MAKAO

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

This demo presents MAKAO (Makefile Architecture Kernel featuring Aspect Orientation), a re(verse)-engineering framework for build systems. At its heart, MAKAO 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.

1. MAKAO's architecture

MAKAO[1] ("Makefile Architecture Kernel featuring Aspect Orientation") is an extensible re(verse)-engineering framework for build systems [1]. Its core features are the visualisation, querying, filtering, refactoring and validation of build systems. Figure 1 shows MAKAO's architecture.

1.1. Build model

Most build tools are inspired by "make" [3]. Internally, they all share a similar Directed Acyclic Graph (DAG) model to keep track of dependencies between the targets from which the system is constructed. MAKAO focuses on a particular build's extracted DAG and links this with static build script data, but not yet with configuration scripts.

DAGs can be easily extracted from the build's trace file. By using the Bash shell"s "xtrace"-option even implicit dependencies can be recorded. These are files which are not explicitly declared as dependencies of a target T, but are used as an argument of the commands building T.

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

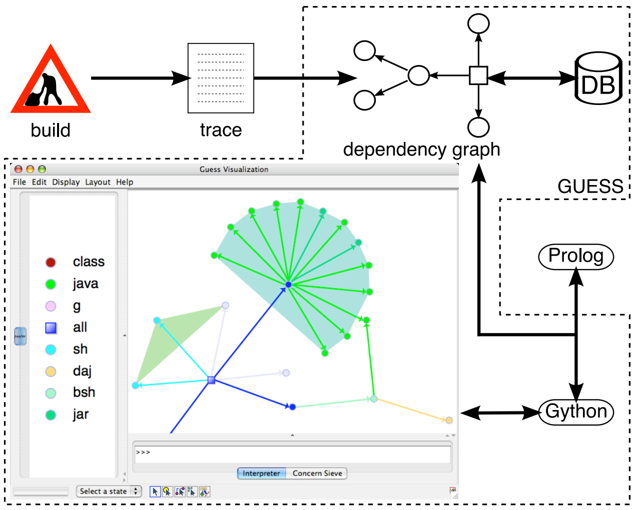

Figure 1. Overview of MAKAO's architecture.

1.2. Underlying technology

Having obtained a build's dependency graph (Figure 1), we want to analyse it. MAKAO uses GUESS[2], which is a graph exploration tool with an embedded Jython-based scripting language called Gython. Graphs can be loaded from file or from an embedded database. All nodes and edges are objects with their own user-definable attributes (name, concern, line number, etc.), which can be manipulated freely to query, navigate or modify the graph. One can execute scripts or invoke some commands in the scripting console. MAKAO extends GUESS through a number of Gython scripts.

To do more powerful querying and filtering of the dependency graph, we prefer a declarative, rule-based approach. For this, we integrated GUESS with a Prolog engine, in which an equivalent logical representation of the graph model is stored.

The screenshot in Figure 1 shows a dependency graph

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

in MAKAO. Every target has a color based on its concern, i.e. the type of file indicated by its extension. Unknown concerns are colored light purple by default. Edges have the same color as their destination node. The colored polygons are convex hulls, which gather all nodes having a common value for some attribute, in this case the makefile in which a target is specified.

2. Benefits

We have applied MAKAO in depth on the Linux 2.6.16.18 kernel and on an industrial C system [1]. Furthermore, we have extracted DAGs from GCC 4, Qt4, etc.

Visualisation — MAKAO visualises the *whole* build system, enabling maintainers to quickly assess the system's complexity, composition, build concerns, etc. Zooming and dragging help to get a clear view of important subgraphs.

Querying — Gython commands make querying for build target information, name clashes, tools in use, build directories, etc. straightforward. By selecting various targets based on the build concern, dependencies, etc., the visualisation can also be filtered.

Filtering — Logic rules in the Prolog engine can achieve more advanced filtering of redundant build information by removing, coalescing, etc. nodes and edges. We can define new views of the build, e.g. to abstract away low-level details of some build idiom, to generate a build-time view [6] or even to recover the design of the source code.

Refactoring — We opted for an aspect-oriented [5] build refactoring approach. Basically, join points are moments during a build where one would like to execute extra behaviour (advice) like e.g. new commands, extra dependency checks or new rules. A pointcut selects all relevant join points where advice should be woven during the build. Weaving makes advice execute at the right moments, and could happen both logically (inside MAKAO) or physically (changing the actual build scripts).

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.

Validation — The Prolog rules enable an approach in which idioms, error patterns, etc. can be modeled and checked on the build system. Possible uses are finding dead source code or even validation of a refactoring.

3. Existing tools and techniques

There are various categories of related work, but especially querying, refactoring and validation are largely unexplored.

The Build Time View (BTV) Toolkit uses an instrumented version of "make" to generate build time views, i.e. a *visualisation* of the high-level architecture of build systems. Many reverse-engineering tools store data extracted from source code, object files, etc. as facts and relations which can then be *filtered*, reduced and composed (using human intervention) into a high-level architecture of a system. Only Dali [4] (now called: ARMIN) exploits build-related facts for this next to source code.

In [2], the build system of C/C++ systems is *refactored by restructuring the source code* (include dependencies) using a modified GCC, which dramatically speeds up builds.

Remake is an improved GNU Make with extra tracing and a *debugger*. "gmd" is another "make" debugger. 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. Vizant is a similar tool for Ant files. 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.

4. Conclusion

MAKAO offers a flexible DAG model of a dynamic build, which can be queried and filtered imperatively (Gython) or declaratively (Prolog). For refactoring, we chose an aspect-oriented approach combined with logic rule-based validation.

References

[1] B. Adams, K. De Schutter, H. Tromp, and W. D. Meuter. Design recovery and maintenance of build systems. In *ICSM '07: Proceedings of the 23rd International Conference on Software Maintenance*, Paris, France, October 2007.

[2] 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.

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

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

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[6] 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.