

Developing and Applying Custom Static Analysis Tools for Industrial Multi-Language Code Bases

Dennis R. Dams¹, Jeroen Ketema¹, Pepijn Kramer², Arjan J. Mooij¹ and Andrei Rădulescu²

¹ESI (TNO), Eindhoven, The Netherlands

²Thermo Fisher Scientific, Eindhoven, The Netherlands

Abstract

Maintaining large, multi-language code bases is challenging because of their size and complexity. Hence, tool support is desirable. Unfortunately, off-the-shelf tools fall short by aiming for genericity instead of exploiting characteristics of the specific code bases and maintenance tasks. Our objective is to support software maintenance by facilitating the development of custom tools for static code analysis.

We report on a case study in which we developed and applied a custom static analysis tool to verify 2441 build dependencies between Visual Studio projects with C++ and IDL code.

1. Introduction

Embedded software of advanced industrial products often consists of large, multi-language code bases that reflect not only the complexity of the product but also the accumulated effect of decades of development. Maintaining the software is challenging due to its size and complexity. Empirical evidence [1] also indicates that multi-language software development is problematic for program understanding.

Off-the-shelf analysis tools often fall short by aiming for genericity instead of exploiting characteristics of specific code bases and maintenance tasks. In our experience, customization is key to getting useful results (cf. [2, 3]). Even general software maintenance tasks may require custom analysis tools due to the use of technologies that are either developed in-house or do not come with good tool support. Our objective is to support software maintenance by facilitating the development of custom tools for static code analysis.

We present an exploratory case study [4] in which we developed and applied custom static analysis tools to analyze build dependencies between Visual Studio projects with C++ and IDL code (see Sect. 3). Initially, we developed the custom tools for a specific code base. Later on, we investigated the reusability on another code base (see Sect. 6).


Our analysis is split in a *model extraction phase* and a *model analysis phase* (cf. [5]). Our model (or knowledge base) is represented as a directed graph (cf. [6]), and contains information on high-level concepts that developers use to reason about their code; lower-level concepts used

BENEVOL'21: The 20th Belgium-Netherlands Software Evolution Workshop, December 07–08, 2021, 's-Hertogenbosch (virtual), NL

✉ dennis.dams@tno.nl (D. R. Dams); jeroen.ketema@tno.nl (J. Ketema); pepijn.kramer@thermofisher.com (P. Kramer); arjan.mooij@tno.nl (A. J. Mooij); andrei.radulescu@thermofisher.com (A. Rădulescu)



© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

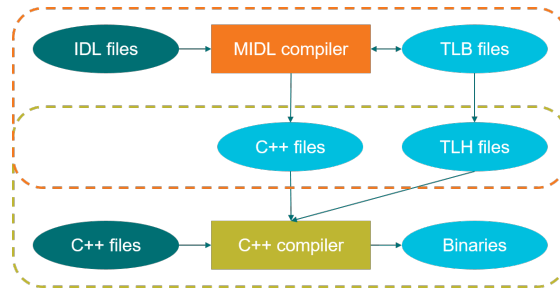


Figure 1: Interplay between IDL and C++

by compilers are omitted. Our main contribution is a *compositional approach* to multi-language and multi-archive model extraction (see Sects. 4 and 5). The results obtained by applying the developed custom static analysis tools are presented in Sect. 7.

2. Preliminaries

We discuss Visual Studio projects and Microsoft’s Interface Definition Language (IDL), as far as relevant to our case study.

Visual Studio Projects Visual Studio projects form the basis of the Visual Studio build system and contain information on dependencies on other projects, files to compile, libraries to link against, files that will be generated (source files and binaries), and compiler flags.

A build dependency occurs when building one project results in the generation of a source or binary file needed to build another project. When one project has a build dependency on another, the developer should declare this.

IDL The Component Object Model (COM) is an interface technology for software components [7]. To define interfaces, IDL is used. IDL language elements include *libraries*, which contain other (non-library) elements, *interfaces*, which are collections of method signatures, and *coclasses*, which are objects implementing interfaces. Libraries, interfaces, and coclasses are identified by UUIDs (Universally Unique Identifiers). IDL also has a `cpp_quote` construct that allows for inserting C++ code fragments into the C++ files generated from the IDL files.

Given an IDL file, the Microsoft IDL compiler (MIDL) generates both C++ files and a type library binary (TLB). TLBs can be imported in C++ and IDL files via, respectively, the `#import` and `importlib` statements. When the C++ compiler encounters a `#import`, it generates a type library header (TLH) based on the imported TLB and `#includes` that header for further processing. Fig. 1 depicts the interplay.

3. Industrial Application: Motivation

Our use case was driven by a desire for software architectural improvements and removing technical debt. Architectural improvements are an enabler for incremental and distributed

builds, assuming correctly specified dependencies. Dependencies can be incorrectly specified in one of two ways [8]:

- *under-declared* dependencies are needed but undeclared dependencies, and may lead to builds that fail occasionally due to missing files (depending on the build order);
- *over-declared* dependencies are unneeded but declared dependencies, and may restrict the build order unnecessarily, reducing build performance, or may prohibit the build altogether (due to dependency cycles).

Verification of dependencies can be time-consuming [9]. Several tools exist that visualize dependencies between Visual Studio projects [10, 11, 12, 13], but these do not verify correctness.

Analysis To verify correctness, we need to compare declared and actual build dependencies. To identify actual dependencies, we look for evidence. A build dependency between two projects is evidenced by a file that is generated by one project, and used by another project. In turn, the need for a file is induced by another kind of dependency, which can also be over- or under-declared. This latter dependency is evidenced by element usage: a dependency on a file is only needed when an element defined in the file is used. Thus, we desire checking the consistency between declared build dependencies, used files, and used elements.

Knowledge Base To perform the analysis, our knowledge base will need to contain several types of relations. Some of these can be extracted from project files, e.g., build dependency declarations, or files needed, compiled, and generated. Other relations can be extracted from C++ and IDL files, e.g., `#include/#import` relations, or the definition and use of elements. Hence, we need to handle multiple languages and combine information from various build stages.

4. Compositional Model Extraction

We next describe our compositional approach to multi-language and multi-archive model – or knowledge base – extraction, as summarized in Fig. 2. Each extraction block in the figure represents an extraction for a single language applied to a single archive. *Merge Graphs* and *Semantic Linking* integrate the results step-wise into a knowledge base. Finally, *Finalize* represents a finalization step that removes information that is used during integration, but is unneeded otherwise.

Single Languages and Archives At the single language and single archive level, we extract information from each individual file. As seen in Fig. 2, some of the extraction blocks also yield information that is fed into other extraction blocks, e.g., *Extract Project* yields a list of C++ and IDL files that is used to drive the extraction in *Extract C++* and *Extract IDL*.

Multiple Languages and Archives *Merge Graphs* takes the (non-disjoint) union of the graphs extracted during the earlier stages, while *Semantic Linking* adds cross-language edges (cf. [14]). The cross-language edges represent the higher-level relations that developers use to reason about their code. For example, relations between uses in C++ files – via TLBs or generated C++ files – of elements whose definitions find their origin in IDL files (see Fig. 1).

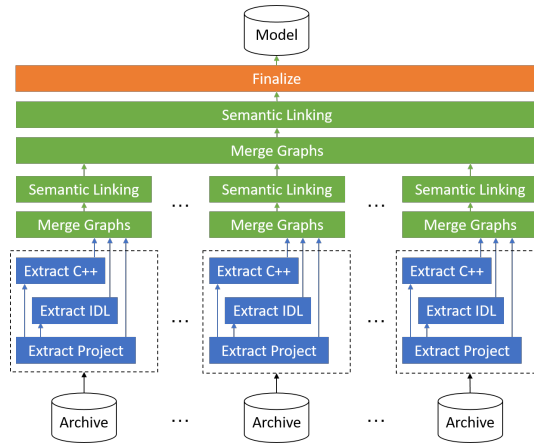


Figure 2: Compositional model extraction

Finalization Extraction also yields information that is used for integration but not needed for our maintenance task. When integration has been completed, finalization removes the additional information, which ensures we do not have to deal with it during analysis. For example, some links to C++ files generated from IDL files are removed, which are no longer needed, as links to the IDL files have been added.

As mentioned, we remove information from the knowledge base that is not needed for our maintenance task. This means, e.g., that we do not include full parse trees in our knowledge base, as they contain a lot of information that we generally do not need. We also take an iterative approach with regard to deciding what information to include in the knowledge base, as we generally do not know beforehand what information is needed for our maintenance task.

5. Model Extraction Challenges

Our main observation is that it is non-trivial to get the extraction details right, mostly due to information originating from different sources. However, we did notice that imperfect approximations can already provide useful information that is not easily available otherwise.

Developing custom tools is both a challenge and relief. It is a challenge when trying to cover everything in full generality. It is a relief, as the customized context does not always require full generality, and provides opportunities to exploit specific knowledge of the considered code.

5.1. Finding Appropriate Parsers

We discuss some specific concerns regarding parsing.

C++ Although commercial C++ parsers exist, we cannot readily experiment with them. Furthermore, open source parsers usually do not handle all C++ dialects we encounter. For this reason we select a parser with good error recovery, like Eclipse CDT [15]. This parser is also used by [16, 2, 17], and, additionally, offers reference resolving [17].

Visual Studio Projects We initially developed our own parser. As project interpretation turned out to be difficult, we migrated to the APIs offered by MSBuild (the tool driving the build in Visual Studio). Although our custom parser was instrumental in moving forward during the early stages of our case study, we consider the use of MSBuild to be preferred.

IDL No open source parser exists that can parse IDL. Hence, we wrote our own. What makes extraction from IDL tricky is the `cpp_quote` construct (see Sect. 2). In our case, the use of `cpp_quote` was limited to constant definitions, where it was desired to link the definitions with their uses in C++ code.

5.2. Unique Naming Schemes

As our knowledge base is a flat graph, we have to uniquely name nodes. A proper naming scheme also simplifies merging multiple graphs. Our scheme is based on the following, where names are prefixed with a node type to avoid name clashes between elements with identical names (such as archives and symbols, or C++ definitions and their (forward) declarations):

- file paths relative to the root of the archive (e.g., for files);
- names (e.g., for archives, symbols);
- UUIDs (e.g., for IDL interfaces, coclasses, libraries);
- hierarchical names relative to files (e.g., for C++ elements, IDL data types);
- hierarchical names relative to UUIDs (e.g., for IDL data types);

The scheme is stable under small changes of the code base, which is convenient when continuing or rerunning an analysis after updating the extracted models. Stability is also useful for assessing code changes by comparing the graphs extracted before and after changes.

5.3. Binary File Formats

To extract information from (generated) binary files (such as TLBs), we can either (1) decompile the binary, (2) use available APIs to read the binary, (3) use textual artifacts derived from the binary during compilation, or (4) use the source code from which the binary was constructed.

We use the third approach for TLBs imported in C++. To this end, we first build the considered archive, generating all TLHs (see Fig. 1), and then run our extraction on the TLHs.

We use the fourth approach for TLBs imported in IDL files, as no textual artifact are generated. The sources of the TLBs may become available when extracting other archives. Hence, we store extra information on IDL files and element uses, which is removed during finalization.

5.4. Dealing with the Preprocessor

To be able to properly parse C++ and IDL files, the C/C++ preprocessor needs to process the files. The result of preprocessing may depend on compiler settings. Even for a simple analysis question such as showing the `#include` relations between files, we can either (1) analyze the code for a specific build configuration, or (2) analyze the code for all build configurations. We use the first option, as it provides enough information for our maintenance task.

Applying the preprocessor may expand simple, well-recognizable macros into complicated code fragments. This may confuse developers, as they often reason about macros as they do about functions. Hence, we include macro definitions and all references to them in our model.

6. Model Extraction Reusability

As mentioned in Sect. 5, customized model extraction provides opportunities to avoid hard problems. However, this may complicate reuse. To evaluate this, we applied our model extraction tool to a Visual Studio code base from a different company. We discuss the observed differences. Insight in these helps to separate generic and specific aspects of the extraction.

Build Infrastructure Both code bases have their own custom build infrastructure on top of Visual Studio projects. The configuration files from these infrastructures are easy to parse, but specific to the code base.

File Locations The code bases use different conventions for the folders containing files shared between projects and archives. However, in the case of sharing between archives we do observe that the shared files always live in the same folder relative to the root of the archive, which means that multi-archive merging can be kept generic.

Code Patterns The code bases use COM differently in relation to the two paths through the middle layer of Fig. 1, which we both support:

- each IDL file is compiled by one project, which generates a TLB that can be imported by other projects;
- each project that wants to use a IDL element from a given IDL file compiles that IDL file.

In the case of `cpp_quote`, the code bases use slightly different patterns to define constants. These are typically generic, but occasionally depend on code base-specific macros.

7. Industrial Application: Results

We describe the model analysis phase and the results obtained for the case from Sect. 3. We illustrate the general line of the analysis and do not aim to be complete. We focus on the part of the knowledge base represented by the schema of Fig. 3, where projects p are related by:

- $\text{ProjectDependsOn}(p, q)$: p declares a build dependency on project q ;
- $\text{MidlGenerates}(p, f)$: p invokes the MIDL compiler generating a TLB or C++ file f ;
- $\text{Compiles}(p, f)$: p compiles file f .

The other relations relate to IDL and C++ files f :

- $\text{Includes}(f, g)$: f #includes an IDL or C++ file g ;
- $\text{Imports}(f, t)$: f #imports a TLB t ;

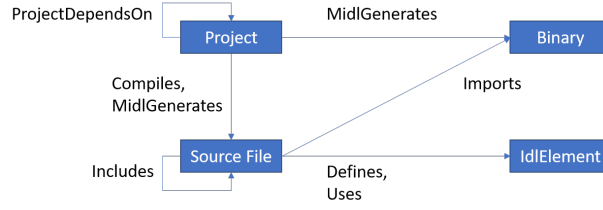


Figure 3: Partial schema of the knowledge base from our case study

- $\text{Defines}(f, e)$: f defines an IDL element e ;
- $\text{Uses}(f, e)$: f uses an IDL element e .

The relations can be combined via composition ($;$), inversion ($^{-1}$), reflexive closure ($?$), and reflexive, transitive closure ($*$).

7.1. Verification Rules for Dependencies

Like [8], we first consider under-declared dependencies. Unlike [8], we distinguish file- and element-level analyses.

Declared Dependencies vs. File References Declared build dependencies should be consistent with file references, which requires relating projects p with generated files f :

$$\text{MidlReferences}(p, f) = \exists q. \text{MidlGenerates}(q, f) \wedge (\text{Compiles}; \text{Includes}^*; \text{Imports}^?)(p, f)$$

Consistency can now be expressed as an inference rule:

$$\frac{\text{MidlReferences}(p, f) \quad \text{MidlGenerates}(q, f)}{\text{ProjectDependsOn}(p, q)}$$

To detect *under-declared* dependencies we read the rule top-down, i.e., if we find a MidlReferences and MidlGenerates pair, we expect a ProjectDependsOn . To detect *over-declared* dependencies we read the rule bottom-up. This latter reading is also used to establish evidence for a declared dependency.

File References vs. Element References File references should be consistent with element references. This requires relating IDL elements e to projects p and files f :

$$\text{ProjectUses}(p, e) = (\text{Compiles}; \text{Includes}^*; \text{Uses})(p, e)$$

$$\text{MidlDefines}(f, e) = (\text{MidlGenerates}^{-1}; \text{Compiles}; \text{Includes}^*; \text{Defines})(f, e)$$

When a project declares a dependency on a MIDL-generated file, it is expected that some file from the project uses an IDL element from the generated file. This is captured by:

$$\frac{\text{ProjectUses}(p, e) \quad \text{MidlDefines}(f, e)}{\text{MidlReferences}(p, f)}$$

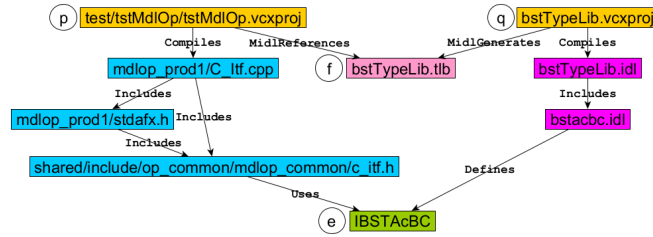


Figure 4: Evidence for the combined dependency analysis

Combining the Rules We can relate declared build dependencies and element references as follows:

$$\frac{\frac{\text{ProjectUses}(p, e) \quad \text{MidlDefines}(f, e)}{\text{MidlReferences}(p, f)} \quad \text{MidlGenerates}(q, f)}{\text{ProjectDependsOn}(p, q)}$$

An under-declared dependency evidenced by this rule is shown in Fig. 4, where p and q are linked via f and e , but where no ProjectDependsOn edge exists. An over-declared dependency may, e.g., be the result of a developer not removing a project dependency after having removed the use of an IDL element.

7.2. Implementation and Results

Based on the above rules and several others, we developed *Depanneur*, a tool for analyzing build dependencies. The tool queries the graph created during model extraction, and produces both textual output and pictures like the one in Fig. 4.

The code base we considered consists of about 1080 Visual Studio projects with 2441 build dependency declarations (before any fixes). *Depanneur* reports 498 under-declared dependencies via MIDL-generated files. This relatively high number may point to over-declared MidlReferences , where no elements are actually used. We corrected all under-declared dependencies by adding them using a custom tool.

As noted by [8], the removal of over-declarations is best done after fixing under-declarations to avoid a temporary increase of build failures. *Depanneur* reports 622 over-declarations. Based on our discussions with the developers, these indeed seem to be over-declarations.

8. Related Work

Overviews of static analysis techniques for multi-language code bases can be found in [14, 18]. A generic approach to cross-language analysis and refactoring is described in [14]; as in our case, language-specific meta-models are used. Our extraction approach resembles that of [19] for finding JNI dependencies between Java and C/C++. First, languages are treated independently; only later is integration considered.

An overview of dependency analysis techniques can be found in [9]. Some more recent approaches are [20, 21, 8, 22].

The Bauhaus Tool Suite [23] focuses on typical kinds of program analyses and reverse engineering, with professional services for customer-specific tailoring of the analyses. Our focus is fully on customizability using open source tools.

9. Conclusions

We presented a case study around the industrial challenge of build dependencies. As no off-the-shelf analysis tools were available, we proposed to develop and apply custom tools. To facilitate the development of custom tools, our approach is three-fold: (1) compositional model extraction to handle multi-language and multi-archive code bases, (2) exploiting specific characteristics of code bases, and (3) graph-based analysis.

Acknowledgments

The authors wish to thank Pi erre van de Laar from ESI (TNO) for valuable feedback.

The research was carried out as part of the Renaissance program under the responsibility of ESI (TNO) with Thermo Fisher Scientific as the carrying industrial partner. The Renaissance program was supported by the Netherlands Ministry of Economic Affairs (Toeslag voor Topconsortia voor Kennis en Innovatie).

References

- [1] P. Mayer, M. Kirsch, M. A. Le, On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers, *J. Software Eng. R&D* 5 (2017) 1. doi:10.1186/s40411-017-0035-z.
- [2] D. Dams, A. J. Mooij, P. Kramer, A. Radulescu, J. Vanhara, Model-based software restructuring: Lessons from cleaning up COM interfaces in industrial legacy code, in: 25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, 2018, pp. 552–556. doi:10.1109/SANER.2018.8330258.
- [3] E. Hajiyev, M. Verbaere, O. de Moor, *codeQuest*: Scalable source code queries with datalog, in: 20th European Conference on Object-Oriented Programming, ECOOP 2006, volume 4067 of *LNCS*, 2006, pp. 2–27. doi:10.1007/11785477_2.
- [4] S. Easterbrook, J. Singer, M.-A. Storey, D. Damian, Selecting empirical methods for software engineering research, in: *Guide to Advanced Empirical Software Engineering*, Springer, 2008, pp. 285–311.
- [5] Y. Zhao, G. Chen, C. Liao, X. Shen, Towards ontology-based program analysis, in: 30th European Conference on Object-Oriented Programming, ECOOP 2016, volume 56 of *LIPICs*, 2016, pp. 26:1–26:25. doi:10.4230/LIPICs.ECOOP.2016.26.
- [6] J. Ebert, V. Riediger, A. Winter, Graph technology in reverse engineering: The TGraph approach, in: 10th Workshop Software Reengineering, WSR 2008, volume 126 of *LNI*, 2008, pp. 67–81. URL: <http://subs.emis.de/LNI/Proceedings/Proceedings126/article2088.html>.
- [7] D. Box, *Essential COM*, Addison-Wesley Professional, 1998.

- [8] J. D. Morgenthaler, M. Gridnev, R. Sauciuc, S. Bhansali, Searching for build debt: experiences managing technical debt at Google, in: 3rd International Workshop on Managing Technical Debt, MTD 2012, 2012, pp. 1–6. doi:10.1109/MTD.2012.6225994.
- [9] T. B. C. Arias, P. van der Spek, P. Avgeriou, A practice-driven systematic review of dependency analysis solutions, *Empirical Software Engineering* 16 (2011) 544–586. doi:10.1007/s10664-011-9158-8.
- [10] J. Wilmans, *depcharter*, accessed December 2021. URL: <https://github.com/janwilmans/depcharter>.
- [11] S. Dahlbacka, *dependencyvisualizer*, accessed June 2021. URL: <https://archive.codeplex.com/?p=dependencyvisualizer>.
- [12] J. Penny, Viewing dependencies between projects in Visual Studio, 2009. URL: <http://www.jamiepenney.co.nz/2009/02/10/viewing-dependencies-between-projects-in-visual-studio/>.
- [13] iLFiS, Project dependency graph generator, 2004. URL: <https://www.codeproject.com/Articles/8384/Project-dependency-graph-generator>.
- [14] P. Mayer, A. Schroeder, Cross-language code analysis and refactoring, in: 12th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2012, 2012, pp. 94–103. doi:10.1109/SCAM.2012.11.
- [15] Eclipse, C++ development tools (CDT), accessed December 2021. URL: <http://www.eclipse.org/cdt/>.
- [16] R. Aarssen, J. J. Vinju, T. van der Storm, Concrete syntax with black box parsers, *Programming Journal* 3 (2019) 15. doi:10.22152/programming-journal.org/2019/3/15.
- [17] D. Piatov, A. Janes, A. Sillitti, G. Succi, Using the Eclipse C/C++ development tooling as a robust, fully functional, actively maintained, open source C++ parser, in: 8th IFIP WG 2.13 International Conference on Open Source Systems, OSS 2012, 2012, p. 399. doi:10.1007/978-3-642-33442-9_45.
- [18] Z. Mushtaq, G. Rasool, B. Shehzad, Multilingual source code analysis: A systematic literature review, *IEEE Access* 5 (2017) 11307–11336. doi:10.1109/ACCESS.2017.2710421.
- [19] D. L. Moise, K. Wong, Extracting and representing cross-language dependencies in diverse software systems, in: 12th Working Conference on Reverse Engineering, WCRE 2005, 2005, pp. 209–218. doi:10.1109/WCRE.2005.19.
- [20] LLVM Team, *include-what-you-use*, accessed December 2021. URL: <https://include-what-you-use.org/>.
- [21] B. Cossette, R. J. Walker, Dsketch: lightweight, adaptable dependency analysis, in: 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2010, ACM, 2010, pp. 297–306. doi:10.1145/1882291.1882335.
- [22] P. Wang, J. Yang, L. Tan, R. Kroeger, J. D. Morgenthaler, Generating precise dependencies for large software, in: 4th International Workshop on Managing Technical Debt, MTD 2013, 2013, pp. 47–50. doi:10.1109/MTD.2013.6608678.
- [23] A. Raza, G. Vogel, E. Plödereder, Bauhaus – A tool suite for program analysis and reverse engineering, in: 11th Ada-Europe International Conference on Reliable Software Technologies, Ada-Europe 2006, volume 4006 of *LNCS*, 2006, pp. 71–82. doi:10.1007/11767077_6.