

Making Application Build Safer Through Static Analysis of Naming

Antoine Beugnard^a and Julien Mallet^b

IMT Atlantique/Lab-STICC, Brest, France

Keywords: Build Process, Verification, Heterogeneous Name Resolution.

Abstract: A lot of studies demonstrate that many builds of software fail, due to dependency issues. We make the assumption that failures are caused by the difficulty of tools to check interdependencies in a context of heterogeneity of languages. This article describes a novel approach to improving applications builds safety based on an abstract interpretation of name usage. Since application building relies on very heterogeneous resources and languages, the approach extracts what appears as a common factor: names. We reuse a name dependency approach (scope graph) already used in single language context, and adapt it to a multi-language environment. It allows to check external references and ensure the resolution of names. Thanks to an operational semantics of build operations on scope graphs, the verification can be done statically, prior to any real build run.

1 MOTIVATION

Software engineering is not just about software, it includes building processes. A strong assumption is that software quality depends on process quality (Paulk, 2009). If there are many programming languages for software, there also exist many languages describing processes. We include among them build scripts and build languages.

If programming languages have been scrutinized and complemented with plenty of tools (static and dynamic checkers, bad smell detectors) it is far the case for build languages. Most studies concern dependency computation and execution optimization.

Mokhov et al. made the same observation in (Mokhov et al., 2018) writing “[b]uild systems (such as MAKE) are big, complicated and used by every software developer . . . [but] [t]hese complex build systems use subtle algorithms, but they are often hidden away, and not the object of study.”

As a consequence, literature shows many issues in build execution. For instance, in (Seo et al., 2014) Seo et al. rely on an empirical study of 26.6 million builds over nine months by thousands of developers to analyze, among other things, failure frequency. They observe that 37.4% of C++ builds fail, and 29.7% of Java’s. They assert that “[t]he most common errors are associated with dependencies between components; developers spend significant effort resolving


these build dependency issues”. However, they conclude that they lack quantifiable evidence on the reasons for builds failure.


Causes of failure and their classifications may vary from study to study (Miller, 2008; Kerzazi et al., 2014; Sulír and Porubán, 2016; Vassallo et al., 2017). However, all these empirical studies converge to the conclusion that builds frequently fail.

We intend to improve build safety by reducing the number of failures. Observing the context of builds and reason of failure, we raise the assumption that failures may be due to heterogeneity of languages used and lack of tools checking interdependencies. Not only programming languages, but also configuration, build and script languages used during the development process. Heterogeneity introduces complexity and makes verification difficult. Middleware like CORBA or .NET, made to improve interoperability are complex and do not solve semantics issues (Beugnard and Salah Sadou, 2007).

The purpose of this article is to propose a cross-domain and cross-language means of verification that can be applied to the process of build. The main idea is to use names and to hide any computation details and then semantics subtleties. Names are intensively used to identify artifacts such as files, elements of programs or processes, resources, tasks, and so on. The proposed approach relies on the assumption that dependency failures are due to name misspelling and file misplacement. Naming is error-prone.

In the next section, we introduce prerequisites,

^a  <https://orcid.org/0000-0002-3096-237X>

^b  <https://orcid.org/0000-0001-5068-1754>

then we describe the approach consisting in building abstract interpretation¹ of the build process based on scope graph (Neron et al., 2015; Antwerpen et al., 2016) of names. Then, section 4 applies the approach on a small C program and its *makefile*, organized as a build process. An analysis of the approach is detailed section 5, before the conclusion.

2 PREREQUISITES

Before sketching the approach, we need to introduce the theory of name resolution introduced by Neron et al. in (Neron et al., 2015). This theory has been used to describe, explain and compare many name binding techniques used in different programming languages. We do not expect to summarize all aspects in a few lines, but we intend to provide the minimum concept understanding.

Name organization is described in a graph. There are two kinds of vertices: names (rectangles) and scopes (ovals). There are four kinds of edges: name definition (scope to name), name use (name to scope), scope naming (name to scope with a specific white arrowhead), and scope hierarchy (scope to scope).

Resolving a name consists in finding a path from a name usage to a name definition. This very simple resolution can be specialized and guided by many more information that can be attached to the edges (through edge tags). Moreover, specific rules (such as naming convention) can be checked, stored and reused.

One important idea is that this resolution mechanism is universal and reusable, and rely on the semantic encoded in edges (and their attached information). There is no need for specific or customized developments. Finally, rules can be accumulated and reused.

Scope graph figures (3 to 5) use this notation. Red vertices are unresolved names, green ones are resolved with the green dotted arrow.

3 APPROACH

Building software relies on diversified artifacts: source files (possibly in many programming languages), configuration files, resource files (audio, video, style, ...), databases, scripts, etc. all stored in the file system. One of the few shared features among all these resources is that they are named. One reason is to help human beings understand and deal

¹The computation is reduced to the evolution of name definition and use through scope graphs.

with these things. Moreover, many resources contain themselves references to other named resources.

All these names can be organized as name spaces encoding relationships among these names. The scope graph approach (Neron et al., 2015; Antwerpen et al., 2016) proved its usefulness for analyzing programming languages. We rely on it, in a more heterogeneous environment, but for simpler relationships.

The approach is twofold. First, the abstraction is applied at a time t , on a set of file system resources, and results in a large scope graph where identified names are collected and linked according to their usage (defined or used) and to the nature of the location they are collected (source code, libraries, data files, etc.). For each file, note that the only collected names are names that make reference to an external² information. Other names are assumed to be tackled by specific tools such as compilers. At this stage, it is already possible to check the proper use of names, that is the accessibility of the definition of a name from its location of use. Note that many definitions may be reachable, which is the case of ambiguity or that no definition exists yet, which can be an error or a premature check, the definition of that name being produced (or introduced) later by other operations (or humans).

The second stage is the definition of action semantics on the scope graph for build operations. This second step allows us to avoid applying the previous stage (the construction of the scope graph) each time a verification is required, but permits an incremental evolution of the scope graph. The verification technique used on the initial stage can then be applied on the result.

Figure 1 illustrates both stages. The vertical arrow a denotes the first stage from the real file system to scope graphs, while horizontal arrows, op_i and \widehat{op}_i respectively denote the actual build operation on the file system and its semantics on the abstract interpretation domain. The bottom line shows the rules \mathfrak{R} that have to be checked at some time. The correctness of the approach relies on the assertion

$$\widehat{op}_i(a(FS_i)) \sqsubseteq a(op_i(FS_i))$$

whose interpretation is: valid rules after abstract application \widehat{op}_i on $a(FS_i)$ are included in validated rules after application of the real op_i on actual FS_i .

In the following, the first (initial) stage of the approach is detailed. It consists in 3 steps (See Figure 1 - vertically, reality to abstraction).

a_1 Extract names from resources (including the file system). Names are filtered. For instance, function names in a source code are ignored,

²Outside the file being analyzed.

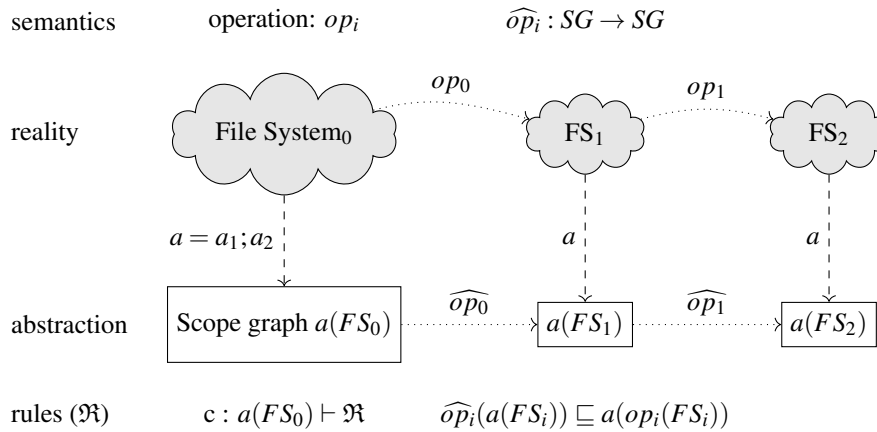


Figure 1: Principle of the abstract interpretation.

since the compiler is supposed to check the well-formedness of the program. This is an adhoc process depending on the nature of the resources. At the moment, we extract names manually; however, automatic extraction has been experimented. Names of interest are constants, file names and other values that denote external resources. An extraction produces a (local/partial) scope graph.

a_2 All (local) scope graphs are gathered in a global scope graph whose backbone is the file system scope graph. This is a simple idea, since the file system gathers itself all other resources³

c A set of rules \mathfrak{R} (bottom of the figure) can be set and checked against the global scope graph. Rules may depend on the stage of the build. They may include good practices such as, this is a example, all videos are in a repository named `videos`. The checker produces a diagnostic on the accessibility of name definitions (*missing*, *correct* - only one, *ambiguous* - more that one).

This process would already improve build verification if applied time to time. But, it needs running the build leading to a dynamic analysis. In order to statically check the build, the second stage of the approach is introduced. It consists in 3 steps (See Figure 1 - horizontally, operation after operation).

1. The semantics (top of the figure) of all build operations (compile, link, cd, move, copy, ...) is defined as scope graph transformations. This is done once, and this is reusable.

³This single file system technique could, in principle, be easily extended to a set of file systems. The global distributed system can define a scope in which each single file system is another scope. The problem of scalability remains, however, if millions of systems are to be analyzed.

2. Each time an operation op_i is realized in the real world, or emulated for verification purpose, its interpretation (a scope graph transformation \widehat{op}_i) is applied on the current scope graph.
3. After each operation, verifications (c) can be run on the resulting scope graph as for the first stage.

Once the semantics is defined, it is then possible to interpret statically (by operation emulation) the build process in order to detect errors early.

The tricky part of the scope graph transformation is the need to introduce time dependencies. Fortunately, the flexibility of scope graphs allows tagging edges with any information. Information concerning name creation or name suppression, for instance, can be introduced as tags.

A major interest of scope graph is the reusability of rules. Once defined, sometimes relying on specific tags, they can be stored in libraries of rules. The checker remains the same.

In the following section a short example is developed as a proof of concept.

4 EXAMPLE

4.1 Simple C Program

In order to illustrate the approach, we extended a very small example that was used as a make tutorial⁴. This application is developed in C. It is composed of a main file `hellomake.c`, a functional file `hellofunc.c` with its header file

⁴<http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/>

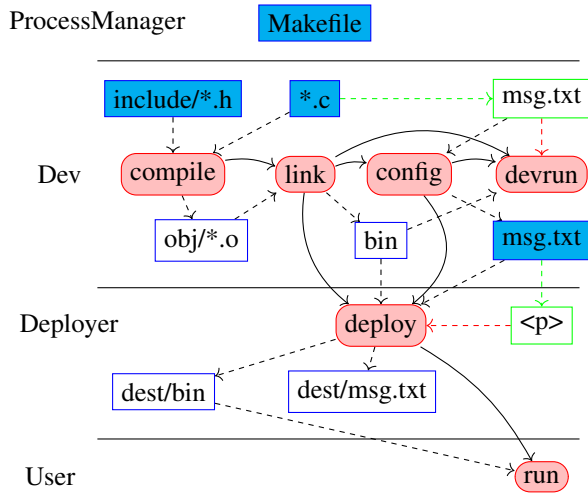


Figure 2: Build process model. Makefile abstraction.

hellomake.h⁵. It also uses an external library stdio.h.

We extended the original application to simulate a more complex development cycle including a configuration and a deployment stage. This application writes in the standard output the content of a file (*msg.txt*). This message is configured before deployment by replacing a pattern (<p>) by an actual value (“!”). Four stakeholders are identified: process manager, developer, deployer, and user. As shown in figure 2, the process manager provides a *Makefile*. The figure is an abstract model of the *Makefile* content. It shows the developer that provides resources (*.*h, *.c and *msg.txt*) then compiles, links and runs to test the program (without configuration); the deployer copies the binary to the target location and configures the application replacing <p> by “!” for instance; then the user runs the deployed and configured application. This simplified process exemplifies that time is essential and that different rules have to be verified at different stage of the development.

This example is not very heterogeneous (only C and text files⁶), but it shows how languages are hidden (abstracted) behind the use of names.

We have written a small scope graph implementation in Java with a checker. Applied to the development directory⁷, the first stage (extraction) gives Figure 3 as the result.

The figure shows the result of a manual extrac-

⁵The name of the header could have been *hellofunc.h*, but the original is named *hellomake.h*: author’s choice.

⁶We do not extract names from the makefile.

⁷For the sake of brevity, libraries and environment variables are ignored.

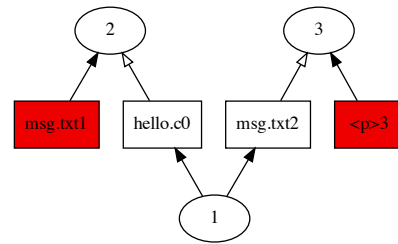


Figure 3: Step 1: scope graph after initial extraction.

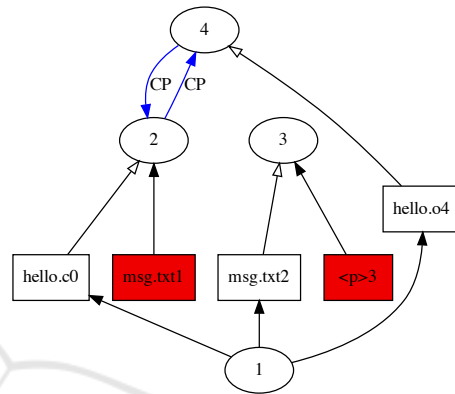


Figure 4: Step 2: after compilation.

tion of names from the files *hello.c* and *msg.txt* which are grouped in a single (partial) file system scope graph. Scopes identifying files (or directories) are round nodes⁸, names are rectangles. An empty arrow head (→) denotes the naming of a scope. In red appear names that are used but have not yet any definition. The node *msg.txt* has not definition because the C file is not compiled neither linked and then has no access to the resource; the node <p> because the configuration is not yet ready.

Figure 4 shows the result of *compilation*. A new file has been created (scope 4 : *hello.o*) which is equivalent from a naming point of view to its source *hello.c*. This is visible thanks to the loop of edges tagged *CP* for compilation. From accessibility of names point of view, nothing changes.

Figure 5 shows the result of *linking*. A new file has been created (scope 5 : *hello*) which is equivalent from a naming point of view to its source *hello.o*. However, since it is executable it has access to the content of its directory. This new property is denoted by the edge tagged *X* (for execution) between the scope 5 and the scope 1. From accessibility of names point of view, now, *msg.txt* is resolved (green). In fact, there is a path from the green box (use) to the location in root (node 1) via *X* edge.

The next step after *compilation* and *linking*, as

⁸The number inside is a simple id that is single.

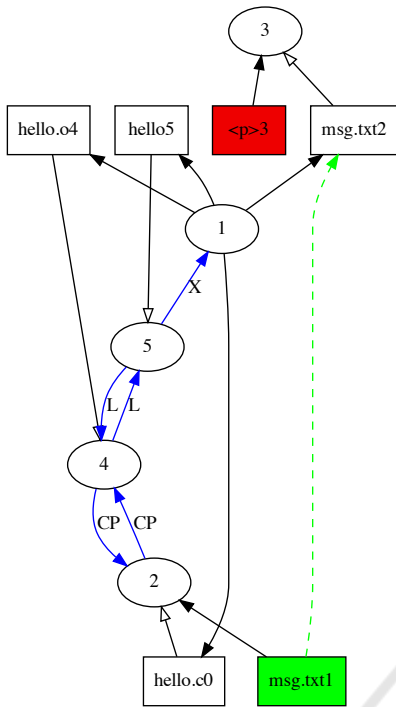


Figure 5: Step 3: after linking.

shown in Figure 2, is *configuration*. Whether the binary would be run (*devrun*), the `msg.txt` file would be found, but the `<p>` would not be configured yet. The configuration is a simple *sed* string substitution in the `msg.txt` file that is triggered by *make*. After the configuration, the scope graph is given Figure 6. Now, a run (*devrun*) would use the configured text. The resolution path goes through the *CF* edge.

The last step is *deployment*. The binary and the `msg.txt` file are copied in the *dest* repository denoted by scope 7. This can be seen (from the name abstraction point of view) on the scope graph, Figure 7. The copied names are also resolved. For instance, `msg.txt9` has a path to `msg.txt14` thanks to the compilation (*CP* 10 to 9), the linking (*L* 9 to 8) and the executability (*X* 8 to 7) of various nodes.

The sequence of scope graphs produced in this example can be obtained after real operations: op_i followed by a (Figure 1). The main advantage of our approach is that these scope graphs can also be produced before any operation, thanks to the semantics of operations on scope graphs: \widehat{op}_i (same Figure). We can then statically verify the proper use of names.

4.2 Semantics

We describe here the semantics of operations that describes how name scopes are composed while the execution of the build is emulated. The semantics of

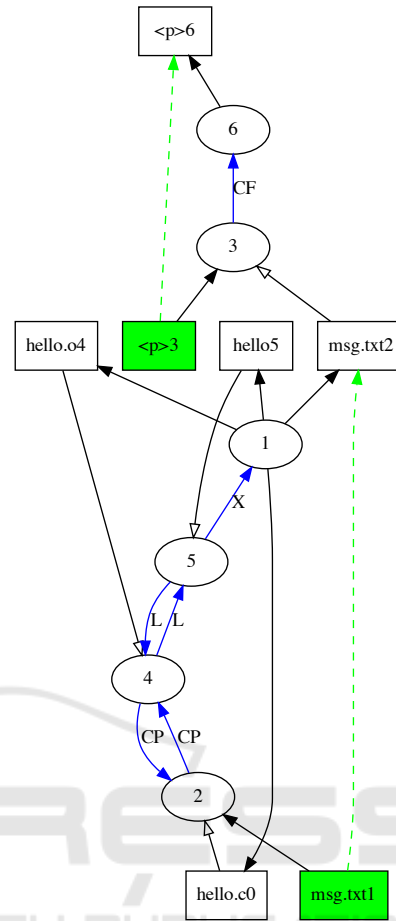


Figure 6: Step 4: after configuration.

operations with respect to scope graphs and naming abstraction is defined Figure 8.

In following rules, sg denotes scope graphs, s scopes. A directory name is denoted by d and a file name by f . o_i denotes operations. An environment env is a couple (s, sg) where the scope s is identified in the sg scope graph as the *current* scope.

(\widehat{Seq}) If o_1 transforms env into env' , the sequence of operations $o_1; o_2$ consumes o_1 and the new state becomes $\langle o_2, env' \rangle$.

(\widehat{Md}) When a directory d is created with `mkdir`, the current scope does not change, but the scope graph is completed with a new link from the current scope to a new name d , and a new scope s_+ named d .

(\widehat{Cd}) If d is the name of a directory linked to the current scope s , then `cd` moves the current directory from s to s_d which is the scope named d in sg .

(\widehat{Compil}) If f is a file in the current directory and s_f is its associated scope, the result of the compilation of f into f' is a new scope s_+ named f' . The scopes s_f and s_+ are bidirectionally linked with the tag *CP*

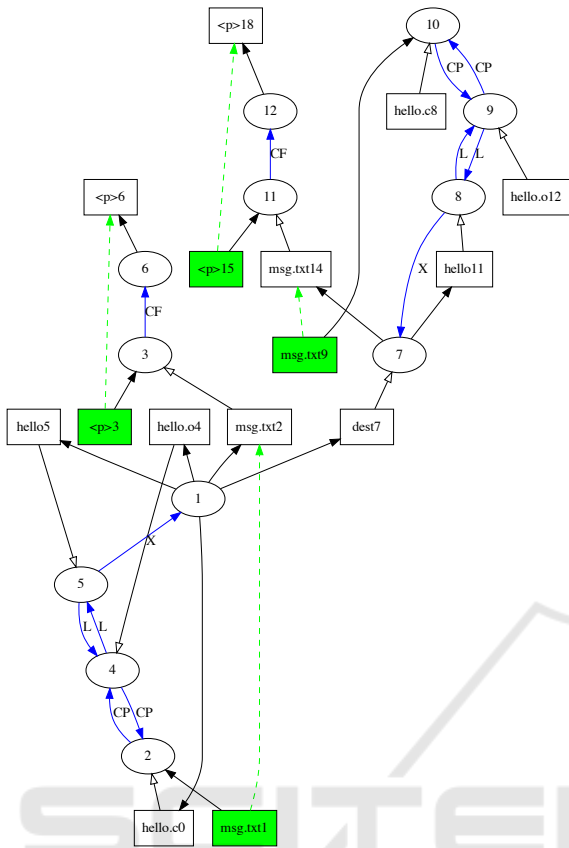


Figure 7: Step 5: after deployment.

(for compilation). This means that the defined and referenced names in the scope s_f are identical in the new scope s_+ . This link introduces an equivalence class containing the scopes s_f and s_+ .

(*Link*) If f_i are files in the current directory, the result of the linkage of all f_i is a new scope s_+ named f' , where s_+ are bidirectionally linked with the tag L (for link) to all original scopes s_i . Another link labeled X (for execution) is introduced between the new scope s_+ and the current directory s . This means that, at runtime, the defined names in s are also reachable from s_+ .

(*Conf - sed*) If f is a file that contains the name x , then the configuration with `sed` does not change the current scope, but complements the scope graph with a new scope s_+ that defines the name x . s_f is linked to s_+ with the tag CF (for configure). This means that the defined name x is reachable from s_f after configuration time.

The semantics of operation `cp` is introduced through two rules according to the executability of copied file.

(*CP1*) If f is the name of an executable ($\overset{X}{\rightarrow}$) file linked to the current scope s and d the name of a di-

rectory linked to the current scope and $\{s_+, sg'\}$ the copy of the sub-scope graph rooted at s_f restricted to CP & L & CF tagged links, then `cp` does not change the current scope, but complements the scope graph with the copy where the new root scope is also named f and f remains executable in that directory (there is a link $\overset{X}{\rightarrow}$ between the new scope s_+ and the destination one s_d). This means that all link introduced through tagged CP, L and CF are copied in order to make reachable the defined names.

(*CP2*) If f is the name of a non-executable ($\overset{X}{\rightarrow} \notin sg$) file linked to the current scope s and d the name of a directory linked to s and $\{s_+, sg'\}$ the copy of the sub-scope graph rooted at s_f restricted to CP & L & CF tagged links, then `cp` does not change the current scope, but complements the scope graph with the copy where the new root scope is also named f . Unlike rule CP1, there is no X link between the copied scope s_+ and the destination one s_f .

These semantics rules have been implemented in Java and where used to produce the previous Figures 3 to 7.

5 ANALYSIS

The proposed approach enables to:

- extract name scope graph from heterogeneous resources;
- build a global name scope graph for a large set of resources with the file system as a backbone;
- define and store rules as an asset for process managers;
- apply verification of rules at chosen steps of the building process;
- check the use of names at the right moment in time. Detect correct use (exactly one path), potential ambiguity (more than one path), or error (no path). This can be done prior to any real execution of the build process;
- produce a diagnostic for each step;
- avoid reconstruction the whole global scope graph thanks to a transformational semantics on scope graph.

The proposed approach is generic and extensible. In order to extend it to new kinds of resources (new programming language, new build language, etc.), you need to implement the name extraction stage for this kind of resource. Further, an extension of the semantics for the new build operations may be required. The abstract naming model (scope graph) and

$$\begin{array}{c}
\frac{\langle o_1, env \rangle \rightarrow env'}{\langle o_1; o_2, env \rangle \rightarrow \langle o_2, env' \rangle} \quad (\widehat{Seq}) \qquad \frac{s \rightarrow d \notin sg}{\langle \text{mkdir } d, (s, sg) \rangle \rightarrow (s, sg \wedge s \rightarrow d \wedge d \rightarrow s_+)} \quad (\widehat{Md}) \\
\\
\frac{s \rightarrow d \wedge d \rightarrow s_d \in sg}{\langle \text{cd } d, (s, sg) \rangle \rightarrow (s_d, sg)} \quad (\widehat{Cd}) \qquad \frac{s \rightarrow f \wedge f \rightarrow s_f \in sg \quad s \rightarrow f' \notin sg}{\langle \text{gcc } -o f' f, (s, sg) \rangle \rightarrow (s, sg \wedge s \rightarrow f' \wedge f' \rightarrow s_+ \wedge s_+ \xrightarrow{CP} s_f)} \quad (\widehat{Compile}) \\
\\
\frac{\forall i, s \rightarrow f_i \wedge f_i \rightarrow s_i \in sg \quad s \rightarrow f' \notin sg}{\langle \text{gcc } -o f' f_1 \dots f_n, (s, sg) \rangle \rightarrow (s, sg \wedge s \rightarrow f' \wedge f' \rightarrow s_+ \bigwedge_i s_+ \xrightarrow{L} s_i \wedge s_+ \xrightarrow{X} s)} \quad (\widehat{Link}) \\
\\
\frac{s \rightarrow f \wedge f \rightarrow s_f \in sg \quad x \in \text{contents}(f)}{\langle \text{sed } -i s/x/e/g f, (s, sg) \rangle \rightarrow (s, sg \wedge s_f \xrightarrow{CF} s_+ \wedge s_+ \rightarrow x)} \quad (\widehat{Conf-Sed}) \\
\\
\frac{s \rightarrow f \wedge f \rightarrow s_f \wedge s \rightarrow d \wedge d \rightarrow s_d \in sg \quad s_f \xrightarrow{X} s \in sg \quad \text{clone}(\{\text{CPL}, \text{CF}\}, (s_f, sg)) = (s_+, sg')}{\langle \text{cp } f d, (s, sg) \rangle \rightarrow (s, sg \wedge sg' \wedge s_d \rightarrow f \wedge f \rightarrow s_+ \wedge s_+ \xrightarrow{X} s_d)} \quad (\widehat{Cp1}) \\
\\
\frac{s \rightarrow f \wedge f \rightarrow s_f \wedge s \rightarrow d \wedge d \rightarrow s_d \in sg \quad s_f \xrightarrow{X} s \notin sg \quad \text{clone}(\{\text{CPL}, \text{CF}\}, (s_f, sg)) = (s_+, sg')}{\langle \text{cp } f d, (s, sg) \rangle \rightarrow (s, sg \wedge sg' \wedge s_d \rightarrow f \wedge f \rightarrow s_+)} \quad (\widehat{Cp2})
\end{array}$$

Figure 8: Abstract semantics of operations.

the verification step apply directly for verifying the build safety. To achieve this, we made the following choices:

- use and reuse of the generic verification algorithm of scope graph that already proof its usability;
- use the scope graph flexibility to adapt the verification mechanism to specific needs in various contexts.

Main limitations are:

- a single file system on a single machine;
- names are constants. Computation on names are not taken into account;
- access right on files are neither taken into account;
- a potentially difficult (and adhoc) extraction of names in unformalized or semi-formalized resources. For instance, Make has no grammar, or configuration files may be just pairs of key-value.

Possible extensions include:

- extending to distributed file systems;
- introducing access right tags, in scope graphs and rules;
- formalizing as contract interfaces the name dependencies among resources. The goal would be to help extraction.

6 RELATED WORKS

We motivate our proposal thanks to many articles on analysis of the build process, see section 1. In that context, an approach to improve build in multilingual context is multilingual abstract interpretation such as in (Mushtaq et al., 2017; Journault et al., 2020; Schiewe et al., 2022), but this work relies on programming language semantics, that we want to hide. Our approach also includes non-programming languages (such configuration files) as the article (Shatnawi et al., 2019) on JEE, including Java, JSP and XML, but that relies on KDM (Pérez-Castillo et al., 2011), a meta-model of resources, less abstract than names. The closest abstraction to ours is the recent work of Ju et al., in (Ju et al., 2023), that focuses on a cross-language name binding. However, they use a deep learning model, not a binding graph such as the scope graph we use. They estimate bindings where we compute them. Finally, in (Zwaan and van Antwerpen, Hendrik, 2023), Zwan et al. tell the story of scope graph that is mainly used for programming language typing and semantics understanding.

7 CONCLUSION AND FUTURE WORK

DevOps operations such as build, configure and deploy require a great deal of know-how. A lot of tools

are available (make, ant, maven, gradle, ansible, chef, etc.). However, and maybe because of the great heterogeneity of languages used, there is almost no formalization of the whole process. We propose here a very high-level abstraction based on name usage to improve the level of trust in DevOps management tools.

Resources offer names to other resources and also require names from them. This can be seen as contracts. These resources are integrated in a process whose stages expect rules to be ensured. These rules are part of the quality process management and could be managed as assets. Operations could then be (partly) specified thanks to pre and postconditions built on these rules.

Beyond the extensions listed in section 5, it should be possible to provide scope graph extractors, operations with their transformational semantics on scope graph, and sets of rules so that a full tooling for DevOps improve the trust in using names. We also envision a background diagnostic process for users of the operating system. Consistency naming rules or good practices could be checked to ensure a better file system management.

We believe that this approach could pave the way for a safer, more formalized, critical activity of software engineering.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their helpful advice in improving this article.

REFERENCES

- Antwerpen, H. v., Neron, P., Tolmach, A., Visser, E., and Wachsmuth, G. (2016). A constraint language for static semantic analysis based on scope graphs. In *the 2016 ACM SIGPLAN Workshop*, pages 49–60, New York, New York, USA. ACM Press.
- Beugnard, A. and Salah Sadou (2007). Method overloading and overriding cause distribution transparency and encapsulation flaws. *Journal of Object Technology*, 6(2):33–47.
- Journault, M., Miné, A., Monat, R., and Ouadjaout, A. (2020). Combinations of reusable abstract domains for a multilingual static analyzer. In Chakraborty, S. and Navas, J. A., editors, *Verified Software. Theories, Tools, and Experiments*, pages 1–18, Cham. Springer International Publishing.
- Ju, Y., Tang, Y., Lan, J., Mi, X., and Zhang, J. (2023). A Cross-Language Name Binding Recognition and Discrimination Approach for Identifiers. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 948–955, Macao, China. ISSN: 2640-7574.
- Kerzazi, N., Khomh, F., and Adams, B. (2014). Why Do Automated Builds Break? An Empirical Study. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 41–50, Victoria, British Columbia, Canada. IEEE Computer Society.
- Miller, A. (2008). A Hundred Days of Continuous Integration. In *Agile 2008 Conference*, pages 289–293, Toronto, ON, Canada. IEEE.
- Mokhov, A., Mitchell, N., and Peyton Jones, S. (2018). Build Systems à la Carte. *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–29.
- Mushtaq, Z., Rasool, G., and Shehzad, B. (2017). Multilingual Source Code Analysis: A Systematic Literature Review. *IEEE Access*, 5:11307–11336. Conference Name: IEEE Access.
- Neron, P., Tolmach, A., Visser, E., and Wachsmuth, G. (2015). A theory of name resolution. In Vitek, J., editor, *Programming Languages and Systems*, pages 205–231, Berlin, Heidelberg. Springer Berlin Heidelberg.
- Paulk, M. C. (2009). A history of the capability maturity model for software. *ASQ Software Quality Professional*, 12(1):5–19.
- Pérez-Castillo, R., De Guzmán, I. G.-R., and Piattini, M. (2011). Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems. *Computer Standards & Interfaces*, 33(6):519–532.
- Schiewe, M., Curtis, J., Bushong, V., and Cerny, T. (2022). Advancing Static Code Analysis With Language-Agnostic Component Identification. *IEEE Access*, 10:30743–30761.
- Seo, H., Sadowski, C., Elbaum, S., Aftandilian, E., and Bowdidge, R. (2014). Programmers’ build errors: a case study (at google). In *the 36th International Conference*, pages 724–734, New York, New York, USA. ACM Press.
- Shatnawi, A., Mili, H., Abdellatif, M., Guéhéneuc, Y.-G., Moha, N., Hecht, G., Boussaidi, G. E., and Privat, J. (2019). Static Code Analysis of Multilanguage Software Systems. arXiv:1906.00815 [cs].
- Sulír, M. and Porubán, J. (2016). A quantitative study of Java software buildability. In *the 7th International Workshop*, pages 17–25, New York, New York, USA. ACM Press.
- Vassallo, C., Schermann, G., Zampetti, F., Romano, D., Leitner, P., Zaidman, A., Penta, M. D., and Panichella, S. (2017). A Tale of CI Build Failures: An Open Source and a Financial Organization Perspective. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 183–193, Shanghai, China. IEEE.
- Zwaan, A. and van Antwerpen, Hendrik (2023). Scope Graphs: The Story so Far. In *Schloss Dagstuhl*, pages 32:1–32:13, Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany. Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann.