Analyzing and Repairing Compilation Errors

Ali Mesbah*, Andrew Rice†, Edward Aftandilian‡, Emily Johnston‡, Nick Glorioso‡,
*University of British Columbia, Canada and Google LLC, USA
†University of Cambridge, England and Google LLC, USA
‡Google LLC, USA

***Keywords*-compilation errors, program repair**

I. Introduction

Resolving a build failure consumes developer time both in finding a suitable resolution and in rerunning the build. Our goal is to develop automated repair tools that can automatically resolve build errors and therefore improve developer productivity.

We collected data on the resolution of Java build failures to discover how long developers spend resolving different kinds of diagnostics at Google. We found that the diagnostic reporting an unresolved symbol consumes 47% of the total time spent resolving broken builds. We found that choice of tool has a significant impact: 26% of command line builds fail whereas only 3% of IDE builds fail. However, the set of most costly diagnostic kinds remains the same for both.

We trained a Neural Machine Translation model on the Abstract Syntax Tree changes made when resolving an unresolved symbol failure. This generates a correct fix with a true positive rate of 50%.

II. Related work

A previous study at Google also showed that many build errors (43%) are caused by issues related to `cant.resolve` [1]. We go further by quantifying the time spent resolving these errors. Program repair has been applied to different domains such as data structures [2], user interfaces [3], and source code of different programming languages including Java [4], [5]. Gupta et al. [6] propose a seq2seq machine learning approach for repairing syntax errors in C. This model is trained on the whole source code of the program rather than just the changes to the AST and achieves a repair rate of 27%.

III. Calculating time to resolve

Every build initiated by a developer at Google is logged. The log contains error messages and a snapshot of the code that was built. We built a custom parser to map error messages in the build log back to the compiler's message templates and therefore to a particular *diagnostic kind*.

We collected Java build logs for two months starting from January 23rd 2018. These were subsequently parsed and analyzed to understand which build errors happen most frequently in practice. We then converted sequences of builds occurring in a particular environment (e.g. for a particular user, in a particular workspace) containing a particular diagnostic into *resolution sessions*.

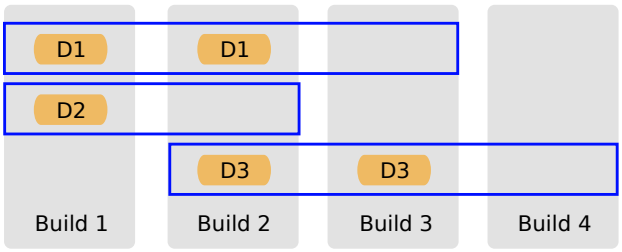

Figure 1. Build Resolution Sessions for four builds with three build diagnostics D1, D2, D3.

A resolution session is a sequence of consecutive builds in which a diagnostic is introduced in the first build, and first resolved in the final build and for which the time difference between successive builds is less than a chosen time window. Figure 1 depicts example resolution sessions for three build diagnostics $D1$–$D3$. The time window captures task switching. If a developer has not performed a build within the window, it is likely they have switched to some other activity, gone to a meeting, lunch, or left the office. We used a value of one hour for this window.

Finally, we calculate the *active resolution cost* (ARC) for a particular diagnostic as the total time between builds in the resolution session divided by the number of diagnostics present in the session. When multiple diagnostics are present we don't know the proportion of time the developer devoted to each one and so ARC shares the time equally between all the diagnostics present.

We found that around 20% (1.0 million out of 4.8 million) of builds were failures. Our logs include a record of the tool used to instigate the build (but not the specific editor used when changing a file). From this we saw that 26% of builds instigated from the command-line resulted in a failure compared with only 3% of builds started from an IDE (IntelliJ or Android Studio).

Table I presents the top ten most costly build errors. In total, there were 1,853,417 compiler diagnostics that were later fixed within a resolution session and 51% (949,325) were `cant.resolve` diagnostics.

We also calculated the relative cost of build errors by

Table I
TOP 10 DIAGNOSTIC KINDS BY ACTIVE RESOLUTION COST (ARC).

Diagnostic kind compiler.err	Description	Active cost (s) Avg	Active cost (s) Max	Builds Avg	Builds Max	Instances Total	Instances %	Cost %
cant.resolve	Use of undefined symbol	69	13327	2.6	143	949,325	51	47
cant.apply.symbol	No method declaration found with matching signature	102	8330	2.6	101	151,997	8	11
strict	Incorrectly declared dependencies in Bazel build system	120	6672	2.2	72	109,156	6	9
doesnt.exist	Use of undefined package	66	7831	2.7	70	159,158	9	7
cant.apply.symbols	No method declaration found with matching signature	115	8771	2.5	41	60,287	3	5
expected	Syntax error	35	5190	2.5	72	168,299	9	4
inconvertible.types	Attempt to cast between inconvertible types	96	6259	2.5	42	38,191	2	3
unreported.exception	Code may throw checked exception, which must be handled	105	4234	2.3	32	22,684	1	2
does.not.override.abstract	Failed to implement inherited abstract method	138	4963	2.8	43	8,089	0	1
already.defined	Symbol already defined	90	5053	2.4	24	12,381	1	1

Table II
DIAGNOSTIC KINDS UNDER DIFFERENT DEVELOPMENT ENVIRONMENTS

	IntelliJ / AndroidStudio 3% of builds fail		Console 26% of builds fail	
Diagnostic Kind	**Instances**	**Cost**	**Instances**	**Cost**
strict	17,255	**32%**	71,232	10%
cant.resolve	22,161	27%	578,058	**47%**
doesnt.exist	10,482	13%	108,450	8%
cant.apply.symbol	7,094	10%	85,844	11%

multiplying the number of diagnostic instances by the average active resolution cost. `cant.resolve` is again the most costly diagnostic kind by far, with 47% of the total active resolution cost. `cant.apply.symbol` is only the fourth most common diagnostic but ranks second in total cost. Despite the large difference in failure rate the set of four most common diagnostic kinds were the same for both console and IDE users (Table II).

IV. REPAIRING `CANT.RESOLVE`

We selected successful resolution sessions with a single `cant.resolve` diagnostic and used a tree differencing algorithm to find the changes that were made to the Abstract Syntax Trees of the source files. We described these changes with a simple language, called DeepDelta, and used these sentences as features for a Neural Machine Translation pipeline [7] with the task of translating from a DeepDelta sentence describing the failure and its location in the AST to a DeepDelta sentence describing the change to resolve it. We extracted 186,992, 42,363, and 36,101 sentences for training, validation and testing respectively with a vocabulary size of $30,000$. We found that the pipeline predicted the correct repair 50% of the time, and that the correct repair was in the top three suggestions 87% of the time.

V. DISCUSSION

A single diagnostic (`cant.resolve`) consumes almost 50% of the overall effort invested in resolving build errors. We found that an off-the-shelf machine translation model does reasonably well at the task of resolving this diagnostic. We expect that more sophisticated modeling techniques [8] should be able to do even better.

Commits to a repository are too coarse-grained to capture resolution of build failures and so we relied on a proprietary dataset for our study. A suitable open dataset would be valuable in this regard.

Performing builds manually from the command-line incurs a much larger cost for resolving build errors and so we plan to investigate the reasons why many software engineers are electing not to use an IDE.

REFERENCES

[1] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' build errors: A case study (at Google)," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2014, pp. 724–734.

[2] B. Demsky and M. Rinard, "Data structure repair using goal-directed reasoning," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2005, pp. 176–185.

[3] S. Zhang, H. Lü, and M. D. Ernst, "Automatically repairing broken workflows for evolving GUI applications," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2013, pp. 45–55.

[4] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2013, pp. 802–811.

[5] F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus, "Automatic repair of buggy if conditions and missing preconditions with SMT," in *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*, 2014, pp. 30–39.

[6] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "DeepFix: Fixing common C language errors by deep learning." in *Proceedings of the Conference on Artificial Intelligence (AAAI)*, 2017, pp. 1345–1351.

[7] Y. W. et. al., "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv*, vol. abs/1609.08144, 2016.

[8] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *arXiv*, vol. abs/1711.00740, 2017.