A Theory of Dual Channel Constraints

Casey Casalnuovo, Earl T. Barr, Santanu Kumar Dash, Prem Devanbu, Emily Morgan
University of California, Davis[1,4,5], University College London[2], University of Surrey[3]
ccasal@ucdavis.edu,E.Barr@ucl.ac.uk,s.dash@surrey.ac.uk,ptdevanbu@ucdavis.edu,eimorgan@ucdavis.edu

ABSTRACT

The surprising predictability of source code has triggered a boom in tools using language models for code. Code is much more predictable than natural language, but the reasons are not well understood. We propose a dual channel view of code; code combines a formal channel for specifying execution and a natural language channel in the form of identifiers and comments that assists human comprehension. Computers ignore the natural language channel, but developers read both and, when writing code for longterm use and maintenance, consider each channel's audience: computer and human. As developers hold both channels in mind when coding, we posit that the two channels interact and constrain each other; we call these *dual channel constraints*. Their impact has been neglected. We describe how they can lead to humans writing code in a way more predictable than natural language, highlight pioneering research that has implicitly or explicitly used parts of this theory, and drive new research, such as systematically searching for cross-channel inconsistencies. Dual channel constraints provide an exciting opportunity as truly multi-disciplinary research; for computer scientists they promise improvements to program analysis via a more holistic approach to code, and to psycholinguists they promise a novel environment for studying linguistic processes.

ACM Reference Format:
Casey Casalnuovo, Earl T. Barr, Santanu Kumar Dash, Prem Devanbu, Emily Morgan. 2020. A Theory of Dual Channel Constraints. In *New Ideas and Emerging Results (ICSE-NIER'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3377816.3381720

1 INTRODUCTION

Code is dual channel: it combines an algorithmic (AL) channel specifying computer execution and a natural language (NL) channel that explains the purpose and context of that execution. The AL channel derives its meaning from the *semantics* of the code; it is computed. Comments and identifier names are primary examples of the NL channel. They do not affect how the program runs; their purpose is to communicate to other humans [12]. For instance, developers use names to avoid the costly cognitive process of mentally executing code: compare the ease of reading a function name correctly called `quicksort` against manually stepping through the algorithm.

The two channels differ in their audiences. The NL channel targets humans. The AL channel targets *both* CPUs, which execute it, and humans, who must understand and maintain it. Because developers hold both channels in mind when coding, we posit the channels are never truly separate. When a developer writes a function, she does not choose cryptic variable names or ignore typesetting; the choices for correct implementation, informative names, and clear and concise writing are all in effect at the same time. The AL and NL channels of code interact and constrain each other, forming *dual channel constraints* (DCC).

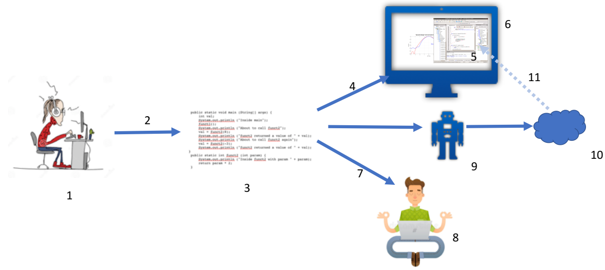

Figure 1: A human developer (1) writes (2) some code (3) on (4) an IDE (5). The code is intended for dual purposes; first, it runs *inter alia*, on a computer (6); second, it is written to read (7) by other humans (8). This human-human channel engenders high-levels of predictability in code, which a machine learner can capture (9) into a model (10); this model, incorporates (11) into the IDE (5) and allows the IDE to leverage its predictions, rankings, and associations to better support the developer (1).

Evidence for DCC lies in both folklore and practice. The precept of good practice, *give identifiers names that reflect their purpose and use*, is just one example of the former. The practice of using coding conventions, such as PEP8 for Python[1] or Google's guide for Java[2], exemplifies the latter. When the audience is other humans, the channels combine to form a human-to-human ($\mathcal{H}2H$) communication medium. To date, research and tools have focused on each channel in isolation and neglected cross-channel interactions. This is a missed opportunity. Programming analysis that ignores the NL channel cannot exploit the hints and signposts within it that may help the analysis scale better, or find more errors more quickly by focusing on code where the two channels have fallen out of sync, *i.e.* in the form of misleading function names or stale comments.

In Figure 1, we clarify how a developer codes in full consciousness of both channels. The AL channel ensures that her code compiles and runs, and can be analyzed within her IDE; she writes her code in ways conducive to readers' expectations across both channels, adding signposts to the NL channel to clarify the AL channel. These code patterns are amenable to being learned by a model, which an IDE can incorporate to assist with coding work.

AL channel constraints on the NL channel also provide a unique environment for studying human communication. The foundational

[1] https://www.python.org/dev/peps/pep-0008t/.
[2] https://google.github.io/styleguide/javaguide.html.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE-NIER'20, May 23–29, 2020, Seoul, Republic of Korea
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7126-1/20/05…$15.00
https://doi.org/10.1145/3377816.3381720

research of Hindle *et al.* showed that code has more repetition than natural language that is exploitable to improve SE tools [10]. More recent work has shown that this repetition is not fully attributable to merely code syntax and grammar; humans must *choose* to write code in a manner more predictable than general natural language [5], and that despite the opportunity to write expressions with equivalent meaning in the AL channel, developers tend to settle on limited options [4]. Thus, there are good reasons to believe that AL channel constraints cause humans to process language decisions *differently.* However, the precise mechanisms that drive these choices are not yet well understood, and a better grasp of them will help both refine linguistic theories of language comprehension and provide insight into writing more comprehensible code.

DCC theory invites multidisciplinary research. For computer scientists, elaborating and tightening a DCC theory not only promises insights into understandable code but also improvements on tasks like defect localization, porting, analysis, and reverse engineering. For psycholinguists, it provides opportunities to test theories about human language comprehension not possible in natural language texts. In short, adopting and studying the DCC theory promises:

- New analyses and techniques that target cross-channel discrepancies and mismatches to find bugs or smells — making a CPU aware of, and able to exploit, the NL channel;
- New prioritization techniques for allocating scarce testing and analysis, both static and dynamic, resources;
- A novel setting, leveraging equivalence in the AL channel, for testing utterance selection theories in linguistics.

2 THEORY AND METHODOLOGY

Languages let humans communicate intended meanings through valid utterances. In natural languages, like English, valid utterances convey meaning through the hierarchical combination of words into sentences, sentences into paragraphs, etc. In programming languages, they can be any program that compiles. Programming languages are (usually) precise and unambiguous. They are also expressive; despite being Turing-complete (capturing any computable function), the same calculation can be expressed in many different syntactic ways. Nevertheless, humans read and write programs, potentially using many of the same cognitive/neural mechanisms as in natural language [7].

Thus, we argue that the $\mathcal{H}2H$ communication channel for code is inherently *noisy*, drawing on similar arguments from Psycholinguistics [14]. This noisy channel has imprecision in transmission. Noisy channel models of language have a long history, but generally these models assume the channel $m \xrightarrow{encode} l \xrightarrow{decode} m'$ where a meaning m from a speaker is encoded into an utterance l and then is decoded by a listener into m', with noise arising from imperfections in both conversions. Moreover, natural language also contains ambiguity, unlike (correct and valid) code. Thus, a listener must utilize as many sources of knowledge as possible—including their full knowledge of the language, their best guesses about the noise model of the environment/transmission process, and their real-world knowledge—in order to probabilistically decode the most likely meaning for the utterance given the context.

While the noisy channel theory is well established for natural language, how does it apply to programming languages? Consider a developer who wants to express some semantic meaning m in a program, implementing it through some syntactic expression ℓ_m. In the $\mathcal{H}2M$ channel, noise nicely captures the developer's imperfect, partial understanding of a program's execution, which encompasses bugs and undefined behavior. In decoding, however, because the semantics are unambiguous given a choice of compiler and platform, there is no possibility of the machine making a "noisy" interpretation of ℓ_m! That is, from the machine 'readers' perspective, *one interpretation* has probability 1.0, and the others are impossible. However, for any given m, there may be several alternative choices for ℓ_m, and the writer will have to choose one; *whatever implementation they choose makes no difference to the machine's ability to make exactly the same interpretation in each case.*

The $\mathcal{H}2H$ channel for code (unlike $\mathcal{H}2M$) is quite different. Developers do not precisely compute the meaning of code like a machine does; instead they use *cues* in the text l_m of the code, to guess the intended meaning m, avoiding if possible the high cognitive load to compute each and every statement, thus giving rise to a condition distribution $p(m' \mid l_m)$ over meanings m' given program l_m. These $\mathcal{H}2H$ cues reside in both NL (l_m^{nl}, denoting comments, variable name choices) and AL (l_m^{al} denoting well-suited, familiar, code construction) components of the code; thus $l_m = (l_m^{nl}, l_m^{al})$. Even with many possible choices of encoding m into l_m, these choices are not equally efficacious as cues on the $\mathcal{H}2H$ channel.

Most programmers, based on their own prior experience, would find certain ways of *writing* code and naming variables better suited and more evocative for certain computations. A capricious, odd implementation choice $\tilde{\ell}$, comprising odd structuring, weird variable names and comments (in $\tilde{\ell}^{nl}$) or unconventional coding choices (in $\tilde{\ell}^{al}$) will very likely vitiate easy reading by another developer, and lower the probability $p(m \mid \tilde{\ell})$; a very clear implementation, with well-chosen variable names, good commenting, and familiar coding forms, $\hat{\ell}$, will yield much higher $p(m \mid \hat{\ell})$.

A programmer, writing code, would, of course, be conscious of this phenomenon - they themselves may read the code later! Given the strong prevalence of coding idioms and style norms, there is good reason to believe these considerations for writing and reading effort will interact and influence the programmer to write code in very conventional ways. In other words, given a meaning m, while the set of choices $\mathcal{L}_m$ to implement m may be quite large, the actual choices $\hat{\ell} \in \mathcal{L}_m$ that programmers pick are quite limited. Thus, the *actually observed* conditional joint distribution of NL and AL constructions $p(\hat{\ell}^{nl}, \hat{\ell}^{al} \mid m)$ for any given meaning m is *highly skewed.* Given m, developers will make specific concurrent choices in code structuring forms ($\hat{\ell}^{al}$) and variable naming and comments ($\hat{\ell}^{nl}$) that interact in expected ways to inform readers, separately and together. Untangling the mutual and independent information in the comments, variable names, and code structures is part of the excitement inherent to the DCC model of code.

Thus far, the theory of the $\mathcal{H}2H$ channel discussed here fits within a large body of psycholinguist literature showing more predictable natural language is both easier to produce and to comprehend [15]. However, as previously noted, code is *more extreme* than (most) natural language in its preference for more predictable forms [5]. This opens up questions about why human cognitive processing of code is in some ways similar to and in some ways

different from human natural language processing. We propose that some of these differences stem from the existence of the two channels operating simultaneously in code. Moreover, code can produce an interesting comparison case with natural language, which will be of interest to researchers in linguistics and cognitive science [7].

3 RELATED WORK

Some software engineering research has explicitly relied on DCC. One example is RefiNym [6]. Some programs suffer from *primitive obsession*, *i.e.* the overuse of simple and underspecified types. Two examples are using strings to hide complicated structures from the type checker or to represent both usernames and passwords. The latter practice prevents the compiler from warning a developer should a password flow into a username. Conceptual types are the types that a developer had in mind while developing, here username and password; program types are the types the developer actually defined and used, here string. RefiNym clusters identifier names that share a program type to recover conceptual types and suggest their use. They call this analysis "bimodal because it intermixes semantics information (flows) and syntactic information (names)" [6]. In this paper, we have intentionally chosen "dual channel" over bimodal because the two channels are distinct and contemporaneous, not different, mutually exclusive states of a single entity.

Casalnuovo *et al.* [4] also invoked an argument of bimodality to understand how code's highly repetitive nature manifests under expressions with equivalent meaning. They transformed expressions into meaning equivalent alternatives, and showed language models could capture developer's preferences controlled by meaning in a cross project environment. Moreover, though the effects vary by transformation type, they persist across programming language, language model, and identifier abstraction. They demonstrated that these language model preferences correlated with human preference via controlled experiment, expanding upon their prior evidence [5] that *human choice* drives much of the repetition in code. However, the question of why human choices might be *more restrictive* in code than natural language remain open.

Several papers have also implicitly used similar ideas to benefit the software engineering community. The code-comment consistency problem, which seeks to find inconsistencies between aligned comment and code pairs, is an inherently dual channel problem. A line of work by Tan *et al.* (*e.g.* [17]) explores this problem. It uses NL techniques to extract topics and rules from comments and compares them with static analysis results to check for inconsistencies. We believe the DCC suggests new directions for solving this problem. (See Section 4). AsDroid [11] mapped user interface text in Android apps to an analysis of what effects (*e.g.* SMS texts) these actions actually performed. Discrepancies between the text and the code enabled them to more efficiently discover malicious behavior.

Finally, one recent paper suggests using a metaprogramming framework to more formally capture "meaning" (in terms of the semantic meaning of classes and methods). The paper provides an example formulation to address the machine's asymmetrical lack information from the NL channel, incorporating both AL and NL information into program analysis [16].

4 IMPLICATIONS AND FUTURE WORK

Dual Channel Constraints provide a compelling explanation for how humans process code and explain code artifacts. We conclude with ways to improve tools by taking advantage of the asymmetrical relationship of the two channels as well as implications for both code and language comprehension.

Implications for Program Analysis. For program analysis and software engineering tools, DCC theory suggests interactions between the NL/AL channels can be exploited to a) find defective and misunderstood code/documentation and b) leverage NL information to prioritize limited analysis resources in AL techniques.

AL/NL channel discrepancies: The DCC theory implies that *discrepancies* between the NL and AL channels strongly indicate that a developer has produced confusing or outright defective code. As developers write in the $\mathcal{H}2H$ medium, DCC require that they maintain both a mental model of how the machine operates and how other developers will interpret it. Since a developer's understanding of both of their "audiences" is imperfect (or noisy), the NL and AL channels can become desynchronized. How might this happen?

The difficulty of mentally simulating computation in the AL channel likely leads to reliance on NL signals reduce cognitive load, and even when fully simulating the AL channel developers may do so imprecisely. Such imprecise mental models of the machine are susceptible to being faulty, even when the developers are highly experienced. For example, consider the Spectre class of security vulnerabilities, which allowed hackers to expose arbitrary memory locations and access sensitive data within an application. These attacks took advantage of *speculative execution*, where the CPU executes branch instructions prior to the results being needed in order to obtain better performance. Relevant to DCC, this attack was possible because even the highly experienced engineers implementing speculative execution did not fully understand the security assumptions in the AL channel other developers relied on [13]. Undefined behavior in C is another case where a developer's mental model of the AL channel may break down, because it violates the usual expectation of unambiguous semantics.

If information from the NL channel can be passed to AL tools, such misunderstandings can be mitigated. Expanding upon the example of the code-comment consistency problem [17], a renewed and more comprehensive focus with modern statistical models of code (e.g. [1]) could provide ways to filter which comments are relevant and explanatory, create probabilistic specifications of code, and better identify bugs. Beyond just comments, identifiers and other NL elements in the $\mathcal{H}2H$ channel can provide warning of potentially buggy code. For example, two regions of code broadcasting similar meaning in the NL channel, but behaving very differently in the AL channel, may indicate poorly structured code.

Analysis of the NL channel can likewise benefit from facts in the AL channel. Limitations in the ability of humans to model program execution extend to the way developers obtain feedback on the correctness of the code. When writing code, developers can get feedback from other developers on the correctness and design, from explicit feedback in code reviews and pull requests, online communications, and more casually simply through conversation. $\mathcal{H}2H$ feedback systems are tuned to the way human minds work; their interactivity and ability to quickly correct misunderstandings

enable faster and more comprehensive understanding. In contrast, messages from the compiler or automated checking tools are static and limited. We believe that combining statistical models that learn from the interactions in human communications and those that leverage the code itself can improve the interactivity and presentation in the error feedback from such tools.

Prioritizing Resources using DCC: Traditional processes for analyzing code, which operate along the AL channel, can provide strong guarantees on soundness, but suffer when applied to real-world code as their abstractions either over-approximate the code (in static analyses), or require execution to provide weaker, under-approximated assurances based on the set of observed runs (in dynamic analyses). Statistical language modeling of code provides a way to limit static analysis over-approximations without running the code, drawing hints from the NL channel. In a non-adversarial setting, names and comments can prioritize the allocation of resources to testing and analyzing the AL channel. For instance, for dynamic analysis, incorporating NL information could help to seed values in fuzz testing and explore likely avenues to find new program states. Alternatively, in static analysis, this information could give probabilistic guarantees, e.g. pointer analysis could raise a warning if a pointer whose name includes "owner" pointed at an object with multiple owners or if the points-to set of a pointer name including "unique" was not a singleton. Moreover, including the NL channel could help reduce false positives common in static over-approximations and produce more useful output to developers.

Implications for Linguistics. A major linguistics concern is "Why do people choose to say things in the way they choose to say them?". An implicit premise behind that question is that there are a lot of ways to verbally convey the same idea. But, in practice, this turns out to be difficult to study in natural language because any change in the wording of an idea probably comes with at least a small change in meaning (maybe at the level of connotations).

Because of DCC, code provides an interesting test case because, in the AL channel, we can frequently identify expressions that are *exactly* equivalent in meaning. So for example, in natural language, people say "bread and butter" about 99% of the time, but 1% of the time they will say "butter and bread". Was the rare ordering chosen to convey something different (*e.g.* a joke along the lines of "Would you like some bread with your butter?") or was it just a mistake? Code provides an interesting comparison case. We could use language models to predict how strongly people should prefer equivalent expressions in natural language ("bread and butter" vs. "butter and bread") and in code (`i+1` vs `1+i`). If we take matched samples of natural language and code (in terms of LM-predicted preference) and find that in practice people deviate more from the LM preference in natural language than in code, that would suggest that the pressure for expressivity in natural language drives people to use the unpredicted form more often. In contrast, if people deviate from LM-predicted preferences equally often in natural language and code, that would suggest that these deviations are not driven by expressivity but merely by the noisiness of human cognition.

More questions arise on whether code's predictability is driven by *production* pressures (e.g. more predictable code is easier for the writer to produce [3]) or by *comprehension* pressures (e.g. readers comprehend more predictable code more easily, and the writer anticipates this [9]). In natural language, these two pressures have proven difficult to disentangle [2, 8]. Code may provide new opportunities for testing this question by providing cases where these pressures conflict—for instance, when a developer's personal preferred style conflicts with the style of a project.

Or, we could simply accept the fact that production and comprehension pressures are often aligned and, rather than trying to disentangle them, treat them as co-present and use their combined force to ask questions about the programmer's cognitive model of predictability (assuming production and comprehension tasks largely share this model). For example, to what extent does whitespace influence how programmers interpret code? Are special characters treated differently than alphanumeric characters?

Finally, we return to asking why code is more predictable than natural language. Is this from the extra cognitive load posed by the programmers' awareness of the AL channel? Or does the explicit, unambiguous semantics of the AL channel discourage programmers from using the small connotative differences possible in alternate phrasings of the same ideas in natural language? Regardless of the answers, the DCC theory provides a framework to address these questions and drive novel and multidisciplinary research.

ACKNOWLEDGMENTS

This work is partially supported by NSF grant 1414172 and EPSRC grant EP/J017515/1.

REFERENCES

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to Represent Programs with Graphs. *arXiv preprint arXiv:1711.00740* (2017).
[2] Jennifer E Arnold, Jason M Kahn, and Giulia C Pancani. 2012. Audience design affects acoustic reduction via production facilitation. *Psychonomic bulletin & review* 19, 3 (2012), 505–512.
[3] Kathryn Bock. 1987. An effect of the accessibility of word forms on sentence structures. *Journal of memory and language* 26, 2 (1987), 119–137.
[4] Casey Casalnuovo, Kevin Lee, Hulin Wang, Prem Devanbu, and Emily Morgan. 2019. Do People Prefer "Natural" code? arXiv:cs.CL/1910.03704
[5] Casey Casalnuovo, Kenji Sagae, and Prem Devanbu. 2018. Studying the Difference Between Natural and Programming Language Corpora. *EMSE* (2018).
[6] Santanu Kumar Dash, Miltiadis Allamanis, and Earl T Barr. 2018. RefiNym: Using Names to Refine Types. In *Proceedings of the 26th ACM ESEC/FSE*. ACM, 107–117.
[7] Evelina Fedorenko, Anna Ivanova, Riva Dhamala, and Marina Umaschi Bers. 2019. The Language of Programming: A Cognitive Perspective. *Trends in cognitive sciences* (2019).
[8] Victor S Ferreira and Gary S Dell. 2000. Effect of ambiguity and lexical availability on syntactic and lexical production. *Cognitive psychology* 40, 4 (2000), 296–340.
[9] Timothy M Gann and Dale J Barr. 2014. Speaking from experience: Audience design as expert performance. *Language, Cognition and Neuroscience* 29, 6 (2014), 744–760.
[10] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *ICSE '12*.
[11] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. 2014. AsDroid: Detecting Stealthy Behaviors in Android Applications by User Interface and Program Behavior Contradiction. In *Proceedings ICSE*. ACM, 1036–1046.
[12] Donald E Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (1984), 97–111.
[13] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *arXiv:1801.01203* (2018).
[14] Roger Levy. 2008. A noisy-channel model of rational human sentence comprehension under uncertain input. In *Proceedings EMNLP*. Association for Computational Linguistics, 234–243.
[15] Maryellen MacDonald. 2013. How language production shapes language form and comprehension. *Frontiers in Psychology* 4 (2013), 226.
[16] Toni Mattis, Patrick Rein, and Robert Hirschfeld. 2019. Ambiguous, Informal, and Unsound: Metaprogramming for Naturalness. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Meta-Programming (META '19)*. ACM.
[17] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /* iComment: Bugs or Bad Comments?*/. In *ACM SIGOPS Operating Systems Review*, Vol. 41.