Embracing Ambiguity

Kenneth C. Arnold
MIT Media Lab, Mind Machine Project
Cambridge, MA USA
kcarnold@media.mit.edu

Henry Lieberman
MIT Media Lab, Mind Machine Project
Cambridge, MA USA
lieber@media.mit.edu

ABSTRACT

Software helps people fulfill their goals, but development tools lack understanding of those goals. But if development tools did understand how software artifacts relate to higher-level intents and goals, they could help developers reuse code, solve problems, and develop systems that are more robust and easier to use. In this paper, we suggest that supporting software development at a stage before concrete formalization is an area of opportunity for software engineering research. We discuss three aspects that are both core challenges and opportunities for this research area: handling ambiguity, understanding human situations, and flexible reflection about failure, and identify research results suggesting that substantial progress can be made on these problems within a decade. We believe that this research will make it easier to develop software that is more broadly useful and robust, even in the face of everyday uncertainty and failure.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*Natural language*

General Terms

Design, Human Factors

1. INTRODUCTION

Humans often communicate their intentions incompletely and ambiguously. When the problem is open-ended and the situation is incompletely understood, imprecise communication helps us work together by attacking a problem from different directions. Since we leave open many ways of accomplishing our goals, others with different mental or physical affordances can help us. Sometimes a single utterance (even a single word: "help!") is sufficient; other times a dialogue is necessary to clarify the intent. And when our helper's actions prove unsuccessful, he/she can often recognize that and try a different approach.

We find the opposite situation when we interact with computers. They require us to communicate with an almost mathematical precision, mostly about structured instructions, and infrequently about intentions. When the instructions are unclear, the computer usually either gives an error message or blindly follows one possible interpretation. And when some action is unsuccessful, often the only course of action is a preprogrammed generic fallback routine.

Yet the huge differences in capabilities between humans and computers suggests that higher-level communication would be even more helpful in communicating with computers than with other humans. And there are glimmers of hope: web search engines do effectively find concrete results from incomplete queries, for example. But much like compilers are often written in the language they are to compile, substantially more humane environments for crafting software will enable more humane interfaces for users.

	informal	**formal**
abstract	casual description	UML specification
concrete	API documentation, comments	code, tests

Table 1: Examples of program representations at different degrees of formality and levels of abstraction

1.1 Formality vs. Abstraction Level

Our thesis is that future programming environments should work with rich mappings between different kinds of program representations–not just abstract to concrete representations but also informal to formal representations. A "formal" representation has a unique and precise semantic interpretation. Table 1 clarifies the distinction between *level of abstraction* and *formality*. By "casual descriptions" we specifically refer to high-level natural language descriptions as in Figure 1. However, loose formalisms and other representations fill in intermediary points between the extremes of the table. Any of these representations could also happen to be incomplete in a particular case; the distinction between abstract and concrete concerns the level of detail of the representation, not how finished it is.

Since connections between abstract and concrete representations have been well studied, this paper will focus on informality. Understanding informal representations and

FoSER 2010, November 7–8, 2010, Santa Fe, New Mexico, USA.

> The main view is a list of conversations. Clicking on a conversation opens it and presents options to reply or forward. ... When composing a message, the user provides a list of recipients, a subject, and the message body; upon clicking Send, the message is sent to the recipients, after a delay of several seconds during which sending can be undone. ...

Figure 1: An informal, abstract representation of an email management environment

relating them to possible formal representations presents a variety of new challenges and opportunities.

1.2 Mapping Between Levels of Description

In this paper, we will start to explore some alternatives to the current style of interaction with software development tools. The central concept is building rich mappings between representations at different levels of formality and abstraction. These mappings are like those of a developer who has been working on a project for some time: they represent how different parts and characteristics of the code relate to the system's observable functionality, high-level structure, and external constraints. These mappings help the developer pinpoint code locations, or even structural decisions, that may be responsible for a bug; they facilitate quickly adapting the software to suit changing requirements; and if a future project has components with similar requirements, they help locate potentially reusable code.

Combined with flexible and robust reasoning capabilities, these mappings will enable software development tools to support the development process before the developer must commit to concrete and formal representations. The impact will be felt well beyond initial development, however; we will highlight benefits in handling novel circumstances and in robustly responding to failure. Some of these ideas are already prototyped or implemented, others are merely conceptual, but all have wide-reaching impact on the development and use of computer software.

2. CHALLENGES AND OPPORTUNITIES

Since computers must execute a precise sequence of instructions, any imprecision or ambiguity in specifying those instructions risks failure or unexpected behavior. Software development processes have been understandably averse to informal representations, going to great lengths to formalize instruction interpretation and avoid possible ambiguity. So any efforts to enable development tools to work with informal representations will face many challenges. But with each challenge comes great opportunity to improve the reusability, flexibility, and reliability of software. We focus here on three challenges and opportunities: handling ambiguity, understanding human situations, and flexible reflection about failure.

2.1 Handling Ambiguity

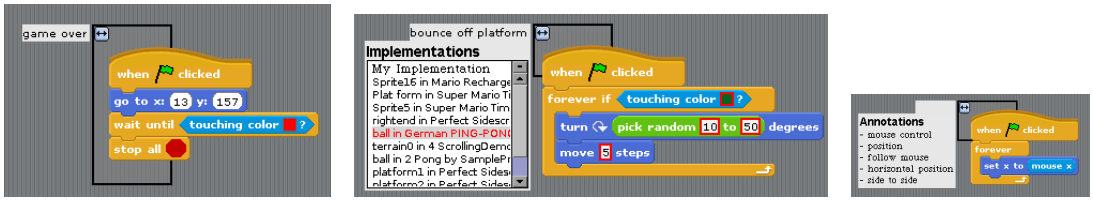
The process of authoring a program can be described as going from ambiguous high-level representations about purpose and approach (mostly contained in the minds of the programmers) to unambiguous, highly structured representations of instructions (mostly contained in computers). Likewise, the process of reading or debugging code can be described as building or verifying a mapping between the concrete code artifacts and abstract intentions and requirements. Development environments provide great assistance for the lowest level of this process: intelligent completion and refactoring greatly aid the input, verification, and manipulation of the highly-structured code representation, and debuggers can help programmers determine why the structured representation behaves differently from their high-level purpose. Also, modern programming languages have gradually freed programmers from less relevant concerns and given them representational tools to guide their middle-level thinking. But most of the process of mapping between levels of description has remained the sole responsibility of the programmer.

Since the end products of programming should be as unambiguous as possible, many programming researchers have been understandably averse to allowing ambiguity even in high-level specifications. However, we suggest that this aversion may be unhelpful because it tends to force programmers to prematurely commit to some particular formal way of thinking before they can dialogue with computers about their programs. It also limits the development tools to only the degree of flexibility afforded by the representation they work with. Instead, we suggest that programming environments could permit programmers to describe desired behavior in more natural terms, even if the terms themselves or the relations between them are ambiguous, then assist the programmer in the task of mapping out how those descriptions relate to unambiguous and concrete software artifacts.

2.1.1 Interactions

A development environment capable of working with relationships between program representations at different levels of detail could help programmers in several ways. First, it could help them find existing code that they could reuse in their programs. The programmer might specify the goal of the immediately desired code, or a subgoal may be inferred from a higher-level goal and the code written so far. Similar techniques could apply to other types of software artifacts as well. For example, a goal could map to both to code and to behavioral tests that (partially) evaluate whether code accomplishes that purpose. And insomuch as failed test cases or other bugs indicate a failure of the concrete code to satisfy the ambiguous goal, we can also think of debugging this way. For instance, we could collect and analyze examples of solutions to problems in certain situations in order to learn what problems might come up in other situations, what their root causes were, and what problem-solving strategies or code changes were helpful in fixing similar problems before. Finally, the programming environment could help the programmer document and distribute the result in a way that permits others to understand and utilize it.

An example of a one-step interaction is our purpose-directed code reuse interface, called Zones [1], shown in Figure 2. Programmers can use a Zone to associate a fragment of code with a natural language description of its purpose. By omitting either the purpose or the code, this annotation interface becomes a search interface, which becomes an annotation when the search is complete. The interaction is thus much like code search, but the search queries can include abstract characteristics that would not match a keyword in an identifier or type, and the retrieved code need not be commented. Also unlike a typical code search interaction, Zones learns how programmers describe purpose by capturing both successful and



(a) A Zone (black rectangle with flag) associates a fragment of Scratch code with a statement of its purpose. The top-left button invokes the search sidebar.

(b) Given a purpose statement, the Zone sidebar (left) shows code that might fulfill it. Selecting an implementation from the list on the left shows its code on the right. As a simulation of future functionality, red boxes surround values that vary among otherwise similar code, highlighting what might need to be changed.

(c) The Zone sidebar can suggest possible purpose statements for a code fragment (simulated for this illustration).

Figure 2: Types of interactions with Zones

unsuccessful search interactions, and simultaneously learns what code characteristics are relevant to that particular goal.

2.1.2 *Promising Research*

Code Search.

One promising research direction is around code search, since search queries can be expressed in a variety of different ways, some more or less ambiguous, and systems can find code within a project or within a code repository that might be relevant to the query. [10] includes a good survey of code search techniques, including formal specifications, type systems, design patterns, keywords, ontologies, tagging, and test cases. A state-of-the-art example of code search integrated into the development process is Blueprint [2]. However, these code search systems have limited ability to reason about purposes that can be accomplished in a variety of ways, and their understanding of natural language is limited at best.

Programming in Natural Language.

Natural language has often been seen as desirable as a high-level specification or programming language because it is a natural medium for communicating goals and ideas with human collaborators. Various attempts have been made to interpret natural language as computer instructions directly, from COBOL to SQL to several modern attempts, including Pegasus [5]. However, since programs must execute unambiguously, many previous attempts at natural language programming have required the use of unnaturally precise wording. Natural language representations of program present many challenges, but we think that managing ambiguity is a core challenge that has not yet received sufficient attention. Several projects have informed our thinking in this regard. Keyword Programming [8] matches keywords to commands and types in a function library. It is a useful tool for managing ambiguity on a low level: when a programmer knows what keywords should appear in a line of code but not exactly how code is formed using these keywords, the Keyword Programming system can use search and type chaining techniques to disambiguate the keyword representation of that line of code. However, the programmer's thinking must still be precise enough to use keywords at the level of a single line of code. Metafor [9] and its successor MOOIDE [7] use sentence structure and mixed-initiative discourse to understand compound descriptions. MOOIDE further showed that general background world knowledge helps to understand natural language input.

2.1.3 *Mapping General to Specific*

To assist in disambiguation, a programming environment must be able to relate ambiguous descriptions to unambiguous representations. One approach is to try various possible realizations of the ambiguous description; this approach has been successful where the ambiguous description straightforwardly defines a constrained space of possible unambiguous representations. Today's large software libraries and open-source code repositories enable a different approach: learning relationships between ambiguous and unambiguous from examples. If a repository contains both code fragments and natural language descriptions of their purpose, the programming environment can learn how characteristics of the code relate to characteristics of their descriptions.

The backend of our Zones system, called ProcedureSpace, is one example of a system that can flexibly map general descriptions to specific implementations. The knowledge about purpose descriptions is incomplete both because they are ambiguous to begin with and because the system has a very incomplete understanding of the natural language itself: the system may not have sufficient knowledge about a particular word used, or the words may be combined in an unfamiliar way. And knowledge about the code is incomplete in that it is generally not known what parts of the code are relevant to accomplishing the goal and what parts are merely implementation details. However, associations between code fragments and purpose descriptions disambiguate each other: ProcedureSpace, in effect, learns about purpose descriptions from the code they describe, and learns about code by studying how people describe its purpose. ProcedureSpace additionally uses background knowledge about how English words and phrases relate to each other to help understand the natural language descriptions. The reasoning approach of ProcedureSpace is not formal logic but an application of a new technique called Blending [4], enabling the easy use of a variety of different kinds of knowledge, so long as they can be made to overlap.

2.2 Understanding Human Contexts

The rapid proliferation of websites and smartphones has brought software into direct contact with many parts of our lives. Social constraints such as privacy and relevance

have had profound impacts on the design of even backend systems (such as image storage). Yet that software, and the tools that help design it, hardly understand the contexts and constraints they are working with. The result is a mismatch between software and the human contexts in which it works; recent outcries over unforeseen implications of Facebook's complex privacy settings underscore the possible severity of this mismatch.

A major challenge in dealing with this mismatch is that human contexts are described in very different terms and at very different levels of detail compared to the programs that interact with them. A formal constraint, such as a simple specific condition that must hold for a user's pictures to be visible, will soon be able to be checked automatically, once verification tools become able to handle today's highly distributed datacenters. But understanding human contexts at their description level will allow not just avoiding specific mistakes but also acting appropriately in a variety of situations.

Resources of commonsense knowledge can help development environments determine what are reasonable or expected actions or events. A knowledgebase represented in formal logic, such as Cyc[6], could help reason about specific situations in detail, while a knowledgebase in natural language, such as ConceptNet[3], could help understand broad issues of intent and context. The Zones/ProcedureSpace system described earlier uses ConceptNet to understand natural language descriptions of code intent.

2.3 Reflecting About Failure

If everything always went as the developers expected, writing explicit logic for all situations would be merely tedious. But if something fails, a system that does not understand what higher-level goals it is meant to achieve will not know how to react appropriately. Developers can give explicit instructions about what to do in certain cases, but enumerating all possible failure scenarios is difficult if not impossible in a sufficiently complex system. So for a system to react appropriately to unforeseen failure, it must know how each of its functions relates to the goals that it is trying to pursue.

When one approach fails, one strategy is to try an alternative approach. Consider a development tool that is aware of multiple ways of approaching a problem and presents them to a developer in incompletely specified form. The developer will probably choose to only flesh out one approach in detail to be used in the typical execution of the program. But the developer may choose to also include selected other approaches in the compiled program. In case of failure, the program could switch to an alternative approach, perhaps in communication with the user or a central problem-solving knowledgebase, or perhaps autonomously. Knowledge of the situation, combined with commonsense knowledge as described in the previous section, could be used to make informed guesses at reasonable choices for any details that the developer left unspecified. This failure-management process could happen entirely in the field, or a forward-looking developer could engage "what-if" scenarios in the lab. While fully autonomous problem-solving is an ideal, this approach is helpful even if it merely suggests possible failure scenarios and potentially reasonable courses of action to the developer for review and further specification.

Some existing systems implement some parts of this vision. But existing approaches are often limited to doing something slightly different with the same data and problem representations. But if the system has a representation of its goal at a sufficiently high level, it could switch to an entirely different approach. For example, a common type of failure in message-passing systems is an unreliable or slow communications channel. So rather than failing, it could order the messages by degree of relevance to the current goals of the destination systems, understanding that message priorities may change as goals change. Or it could even find alternative agents more nearby that may be able to accomplish similar goals. This behavior could be hard-coded for certain scenarios and representations, but to work in general, the system must be able to reason about how capabilities accessible to it are related to goals that are, at least from its point of view, incompletely specified or understood.

3. CASE AND MODEL-BASED METHODS

The points and scenarios described thus far may remind readers of CASE or Model-Based Software Engineering tools and methodologies that have waxed and waned in popularity for the past three decades. These systems attempt to address the difficulty of software engineering by raising the level of abstraction in software descriptions and the degree of automation in "compiling" those descriptions into executable code [11]. However, these systems are still formal, in that their input languages must have precise semantics. Ideally it would be easier for humans to manually disambiguate their thoughts into that input language, then use that language alone as a surrogate for the "real" program, but the results in practice have been mixed. We suggest instead that once software systems can deal with ambiguous presentations of the goals they (or the systems they build) should have, the overall process of crafting specifications, tests, and reliable software can be substantially eased.

4. CONCLUSION

Reliable software will know its goals and have multiple strategies to accomplish them in case something goes wrong. Intelligent software development environments will be able to participate in natural human discourse about the program at all levels of detail. Software will understand human contexts and goals and behave appropriately in them. All of these characteristics will be enabled once software engineering tools and practices embrace the reality of ambiguity, acknowledge its strengths, and help programming teams flesh out a complete description of the software that includes both formal and informal descriptions.

5. REFERENCES

[1] K. C. Arnold and H. Lieberman. Managing ambiguity in programming by finding unambiguous examples. In *Onward! 2010 (to appear)*, 2010.

[2] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer. Example-centric programming: Integrating web search into the development environment. Technical report, CSTR-2009-01, 2009.

[3] C. Havasi, R. Speer, and J. Alonso. ConceptNet 3: a flexible, multilingual semantic network for common sense knowledge. In *Recent Advances in Natural Language Processing*, Borovets, Bulgaria, September 2007.

[4] C. Havasi, R. Speer, J. Pustejovsky, and H. Lieberman. Digital Intuition: Applying common sense using dimensionality reduction. *IEEE Intelligent Systems*, July 2009.

[5] R. Knöll and M. Mezini. Pegasus: first steps toward a naturalistic programming language. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 542–559, New York, NY, USA, 2006. ACM.

[6] D. Lenat. CYC: A large-scale investment in knowledge infrastructure. *Communications of the ACM*, 11:33–38, 1995.

[7] H. Lieberman and M. Ahmad. Knowing what you're talking about: Natural language programming of a multi-player online game. In M. Dontcheva, T. Lau, A. Cypher, and J. Nichols, editors, *No Code Required: Giving Users Tools to Transform the Web*. Morgan Kaufmann, 2010.

[8] G. Little and R. C. Miller. Keyword programming in Java. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 84–93, New York, NY, USA, 2007. ACM.

[9] H. Liu and H. Lieberman. Programmatic semantics for natural language interfaces. In *CHI '05: CHI '05 extended abstracts on Human factors in computing systems*, pages 1597–1600, New York, NY, USA, 2005. ACM.

[10] S. P. Reiss. Semantics-based code search. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 243–253, Washington, DC, USA, 2009. IEEE Computer Society.

[11] B. Selic. Personal reflections on automation, programming culture, and model-based software engineering. *Automated Software Engineering*, 15(3-4), 2008.