INSTITUTE FOR DEFENSE ANALYSES

A Partial Survey on AI Technologies Applicable to Automated Source Code Generation

Francisco L. Loaiza, *Project Leader*

David A. Wheeler

John D. Birdwell

September 2019

IDA Non-Standard
NS D-10790

INSTITUTE FOR DEFENSE ANALYSES
4850 Mark Center Drive
Alexandria, Virginia 22311-1882

The Institute for Defense Analyses is a non-profit corporation that operates three federally funded research and development centers to provide objective analyses of national security issues, particularly those requiring scientific and technical expertise, and conduct related research on other national challenges.

About This Publication

This work was conducted by the Institute for Defense Analyses (IDA) under contract HQ0034-14-D-0001, Task DI-5-4630, "Army Software Marketplace Acquisition Strategy," for Product Lead Army Enterprise Staff Management System (PL AESMS). The views, opinions, and findings should not be construed as representing the official position of either the Department of Defense or the sponsoring organization.

Acknowledgments

Ronald G. Bechtold, Jonathan R. Agre

For more information:

Francisco L. Loaiza, Project Leader
floaiza@ida.org, 703-845-6876

Margaret E. Myers, Director, Information Technology and Systems Division
mmyers@ida.org, 703-578-2782

INSTITUTE FOR DEFENSE ANALYSES

IDA Non-Standard NS D-10790

A Partial Survey on AI Technologies Applicable to Automated Source Code Generation

Francisco L. Loaiza, *Project Leader*

David A. Wheeler

John D. Birdwell

Executive Summary

This work was performed in support of the Army Software Marketplace (ASM) Acquisition Strategy project (DI-5-4630). The Army has "identified the need to reduce costs and delivery time across the enterprise related to software generation, access, management, and sustainment."

This work and paper partly fulfill paragraph 3g of the statement of work in the project description, which states the intent to "evaluate technical options and alternatives … for standing up an enterprise-level Army Application Development Environment (ADE) that supports development for the full range of software platforms…." This paper also is intended as an input to Deliverable 4d, which is "a draft report on the maturity and applicability of options that can support the creation of an Army ADE."

In addition, the survey of emerging technologies with a sufficient degree of maturity and applicability to the workflows that will exist in the planned Army Software Factory (ASF) will inform the governance to be articulated in this phase of the ASM study.

The document provides a partial survey of relatively recent research efforts reported in the literature in the area of automated source code generation using methods from artificial intelligence (AI) in general, and machine learning (ML) in particular. The results of the literature search led to discussions among the IDA subject matter experts. Those discussions are captured in this paper.

For this paper, we use a relatively broad definition of AI that includes ML, as well as a number of specific technologies such as reinforcement learning (RN) and recurrent neural networks (RNN). We narrowed our scope to papers published on or after 2010 in order to meet the fiscal and time limitations of this effort. We are aware that in so doing we may have left out a large body of pertinent previous research, as well as the additional context that can be imparted by reviewing literature over a substantially longer baseline, but our assumption is that older, promising approaches are likely to have been refined and incorporated in more recent works, some of which we cover here. This is an active area of research, and although we have tried to identify all important papers therein, there are likely to be papers we have missed due to the volume of activity and our lack of time. Nevertheless, we believe this is a useful survey of a challenging area.

Generating source code through AI is not a solved problem. Many approaches have worked only in laboratory settings and/or only on small-scale problems. If the need is to

develop industrial-scale software today, then more conventional approaches are indicated in almost all cases.

We identified eight promising research efforts but found the following six to be the most worthy of further research funding:

1. Determining what a program should be from a small set of input/output pairs is hard, even for humans, because a large number of programs could generate that data. However, it has been done. IP approaches have successfully developed small programs, such as inferring string manipulation programs in spreadsheets (an approach that is implemented in Microsoft Excel's "flash fill" function). Microsoft's DeepCoder [Balog 2017] uses IP approaches to learn what programs should look like, which can aid in program generation.

2. Translating natural language into source code is hard, especially since typically natural language has many ambiguities, and background knowledge is important. However, the approach in [Yin 2017] looks especially promising — it simplifies the challenge the underlying algorithm must implement and makes it easier for an RNN to discover the recursive structures.

3. Patch generation to repair existing code is a narrower problem but extremely useful, and MIT's Prophet [Long 2016] has been demonstrated on larger codebases.[1]

4. It is possible to convert natural language to UML class diagrams and then use those diagrams to automatically generate source code stubs and the general program structure (see [More 2012], [Sharma 2015], and [Gulia 2016]). This approach all by itself is somewhat limited — the source code for the methods and functions that make up the body of the code stubs would not be generated by this technique, and a programming-level understanding of what to state in natural language is required. However, this approach could be fruitful in combination with some of the other ASCG techniques discussed in this paper.

5. Converting UI sketches into code is promising — both [Beltramelli 2017] and [Wilkins 2018] demonstrate some useful prototypes, although they are expressly limited in scope.

6. Creating a higher-level abstraction for source code, instead of trying to directly generate text. For example, several of the approaches discussed in this paper use ASTs (see [Yin 2017] [Li 2018]). Such higher-level models should make it easier for learning systems to identify patterns and create syntactically correct code (e.g., ensuring that block ending marks match block beginning marks). Using

[1] See https://www.kestrel.edu/home/projects/apt/.

representations that easily model these higher-level structures should make it easier to use well-known techniques such as RNNs and could possibly make it easier to extend those techniques to deal with the underlying structure of source code.

Several of the additional approaches surveyed emphasize creating a higher-level abstraction for source code instead of trying to generate the scripts directly. For example, several use abstract syntax trees [Yin 2017] [Li 2018]. Such higher-level models should make it easier for learning systems to identify patterns and create syntactically correct code (e.g., ensuring that block ending marks match block beginning marks). Using representations that easily model these higher-level structures should facilitate the use of well-known AI techniques such as RNN and possibly make it easier to extend those techniques to deal with the underlying structure of source code.

Given the ever-growing importance of software in all spheres of modern life, we expect that research in this area will accelerate in the future and that more effective methods will be developed and eventually commoditized.

Contents

1. Introduction....1-1
2. Approaches Categorized as Out-of-scope....2-1
   A. Approaches Categorized as Outside AI....2-1
   B. Approaches That Do Not Generate Source Code....2-4
   C. Approaches Categorized as Programming....2-4
   D. Older Materials....2-5
3. Partial Survey of AI Technologies Applicable to Automated Code Generation....3-1
   A. Automated Generation of Source Code Using AI....3-1
   B. Possibly Related Work....3-15
   C. Representations of Programs....3-16
4. Conclusions....4-1

References....R-1

Acronyms and Abbreviations....AA-1

1. Introduction

This work was performed in support of the Army Software Market Place Acquisition Strategy project (DI-5-4630). The Army has "identified the need to reduce costs and delivery time across the enterprise related to software generation, access, management, and sustainment."

This work and paper partly fulfill paragraph 3g of the statement of work in the project description, which states the intent to "evaluate technical options and alternatives … for standing up an enterprise-level Army Application Development Environment (ADE) that supports development for the full range of software platforms…." This paper is also an input to Deliverable 4d, "a draft report on maturity and applicability of options that can support the creation of an Army ADE."

In addition, the survey of emerging technologies with a sufficient degree of maturity and applicability to the workflows that will exist in the planned Army Software Factory (ASF) will provide the rationale for the governance to be articulated in this phase of the ASM study.

The document provides a partial survey of relatively recent research efforts reported in the literature in the area of automated source code generation (ASCG) using methods from artificial intelligence (AI) and machine learning (ML). The results of the literature search led to discussions among the IDA subject matter experts. Those discussions are reported in this paper.

For this paper, we use a relatively broad definition of AI that includes ML, as well as a number of specific technologies such as reinforcement learning (RN) and recurrent neural networks (RNN). We have narrowed our scope to papers published on or after 2010 in order to meet the fiscal and time limitations of this effort. We are aware that in so doing we may have left out a large body of pertinent previous research, as well as the additional context that can be imparted by reviewing literature over a substantially longer baseline, but our assumption is that older promising approaches are likely to have been refined and incorporated in more recent works, some of which we cover here. This is an active research area, and though we have tried to identify important papers in this area, there are likely to be papers we have missed due to the volume of activity and our lack of time. Ideally, we would examine all the cited works after every search, include all relevant ones, and repeat until we were confident all important works were cited. In this case, every time we found some material, we found other material as well and did not get a sense of closure.

Therefore, we can only call this a partial survey. Nevertheless, we believe this is a useful survey of a challenging and active area of research.

Chapter 2 provides a brief discussion of some technology areas that we decided are not in scope for this paper. Chapter 3 presents the survey itself. Chapter 4 ends the main body with a number of conclusions. The references are lengthy, because they include copious annotations (including some abstracts and commentary).

2. Approaches Categorized as Out-of-scope

To complete this paper in a reasonable time, we had to exclude some approaches and materials, as described in the following sections.

A. Approaches Categorized as Outside AI

There is no universal agreement on what is or is not included in the terms "artificial intelligence" and "machine learning." For example, some practitioners use ML to exclusively refer to statistical techniques and argue that ML is not part of AI, resulting in a narrower definition of AI. It is also possible to define AI broadly enough as to encompass any kind of automation, since any automation could be viewed as doing something "intelligent." If we interpret AI too narrowly, we exclude too much, and if we interpret it too broadly, the term lacks any particular meaning. It's possible to define AI as "whatever we don't know how to do," but this is a useless definition; by this definition, we can never have any success. This variance in definitions makes it difficult to survey technologies that use AI, as it is sometimes difficult to determine what to include or exclude.

For this paper, we excluded some approaches for automated source code generation (ASCG) from our list of AI approaches. This, however, does not mean they are useless, or that others may not include them in their collection of AI techniques. In this chapter, we note a few of these approaches, in part to show we did not ignore them, and in part to show that our conclusions would be different if we included them.

Some of these excluded approaches are as follows:

1. *General-purpose programming language compilers developed using traditional techniques*. Compilers translate information in "source code" (this is typically text, but it could also be graphical) into something that can be executed by a real or virtual machine. Historically, the use of compilers was referred to as "automatic programming."[2] If the term AI is considered broadly enough, then all compilers could be considered part of AI, as compiling is clearly something that requires

[2] David Parnas notes that, in the 1940s, the term *automatic programming* referred to assemblers, whereas in later years, it referred to program generation from languages such as FORTRAN or Algol. As he put it, "*automatic programming* has always been a euphemism for programming with a higher-level language than was then available to the programmer. Research in automatic programming is simply research in the implementation of higher-level programming languages… Of course automatic programming is feasible… The only real question [is] the efficiency of the resulting programs. Usually, if the input "specification" is not a description of an algorithm, the resulting program is woefully inefficient." [Parnas 1985].

intelligence when done by humans. Compilation involves symbolic manipulation and is often implemented by transformation rules; both terms are often associated with AI. That said, traditional compilers are usually not included in the AI literature. There are many possible reasons for this exclusion. For example, compiler input usually is in the form of a very rigidly defined programming language (designed to be compilable), instead of a natural language narrative. In addition, the transformation rules typically do not attempt to capture wider world knowledge, and they do not contain any kind of "learning." In any case, there is a very large independent set of literature on compilation of general-purpose programming languages, and those traditional compilation techniques are excluded from this paper. We do include papers when approaches are used outside that literature on compilation (e.g., if the input is natural language or if ML is used).

2. *Model-driven engineering (MDE) / Model-driven architecture (MDA)*. A number of software systems have been developed using an MDE/MDA approach. [Klein 2015b] defines MDE as "a software development approach that treats models as the primary artifacts created and used by software lifecycle processes…. Typical modeling languages include the standards-based Unified Modeling Language (UML) and Architecture Analysis and Design Language (AADL), as well as proprietary languages such as the Integranova's Model Execution System (M.E.S.)." It is possible to generate large amounts of code using an MDE/MDA approach, though whether or not these outputs will be effective in a particular situation depends on a variety of factors, including contractual factors. When code is generated, it is generated from the models. However, the code generation approach in this case can use essentially the same approaches as compilers, to the point where it is difficult to distinguish them. Such approaches are often not considered part of AI; for example, in [Klein 2015b] the terms "artificial intelligence" and "machine learning" never occur in the paper, and the term "knowledge" always refers to organizational knowledge. It is possible to apply AI within these approaches, but we will only include such papers if there is an additional reason to consider them in scope. For more about MDE/MDA, see [Klein 2015a] and [Klein 2015b].

3. *Computer-Aided Software Engineering (CASE) tools, Interactive Development Environments (IDEs), and traditional code generators*. There are many tools that allow people to enter various representations of software (typically graphical) and generate code. These are often not considered part of AI. Again, they tend to use approaches similar to traditional compilers (if they can even be distinguished from them), and terms like "artificial intelligence" or "machine learning" are usually not used to describe them. For example, the survey of [Rosales-Morales 2015]

does not mention AI except when discussing [Danilchenko 2012]. Here too, we will only include such papers if there is an additional reason to consider them in scope. For an older survey and analysis of such tools, see [Fife 1987]. A much more recent survey and analysis is in [Rosales-Morales 2015].

4. *Formal methods, formal verification, formal proof, and stepwise refinement.* There is a long history of applying mathematics to the analysis and development of software. These involve creating mathematical specifications of what the software should do. These approaches often involve proofs that the software or a model of the software meets those specifications. The proofs themselves often involve tools such as *theorem provers*, which show that a given set of assumptions leads inexorably to a conclusion. However, creating proofs is not the same as creating a useful program, so we have excluded theorem provers from this paper.[3] One approach to handling complex systems is stepwise refinement, where sequences of refinement steps transform higher-level statements into more detailed models or code. Note that this can eventually lead to program generation from higher-level specifications. This approach can be very useful when the goal is highly reliable software, but it requires significant mathematical expertise and is much more difficult than writing a program when the reliability requirements are not as strong. Examples of such work include Kestrel Institute's Automated Program Transformations (APT) work,[4] which builds on ACL2.[5] The Coq proof assistant[6] is also widely used. For surveys about formal methods, see [Almeida 2011], [Black 2016], and [Wheeler 2019]. Most of this work is not widely considered part of AI (although some theorem provers are). In addition, the effort required to use these efforts today means that when people choose them, they are choosing them due to very strong reliability or security requirements, not because they wish to automatically generate source code per se. In particular, there is no expectation in the near term that these approaches will be less expensive or faster at generating code than traditional approaches, as creating the rigorous

[3] The Curry-Howard correspondence (also known as the Curry–Howard isomorphism or equivalence) proves that a proof is a program, and the formula it proves is the type for the program. This correspondence is an important theoretical underpinning in programming language theory and in proof theory. However, this correspondence does not mean that a program generated from a proof is efficient enough for direct use, and what's more, generating such proofs is generally much more difficult than simply writing a program. So although this correspondence is an important theoretical tool, it does not necessarily result in the kind of program generation considered in this paper.

[4] https://www.kestrel.edu/home/projects/apt/

[5] http://www.cs.utexas.edu/users/moore/acl2/

[6] https://coq.inria.fr/

specifications and related information is often challenging and time consuming. Thus, we exclude them in this paper.

B. Approaches That Do Not Generate Source Code

There are a number of systems for teaching a computer what to do that do not generate software source code. This includes some systems for "programming by demonstration" that teach a computer or a robot new behaviors by demonstrating the task. Some systems that are described as implementing "programming by example" also do not generate source code. The goal of interactive task learning (ITL) systems is to enable "intelligent artificial agents or robots [to learn] new tasks through natural interactions with humans" [Laird 2017]. The definition of ITL does not strictly forbid generating code, but there is no emphasis on generating source code as the way to represent learned tasks, and its general emphasis is instead on creating systems that do not require programming at all.

These approaches can be useful, but their learned results cannot be reviewed or modified as traditional source code. In many circumstances, the availability of source code may be irrelevant, as often what is wanted is simply a particular kind of behavior or result. Therefore, this paper excludes approaches that do not generate source code.

C. Approaches Categorized as Programming

Some approaches may be categorized by some practitioners as AI but are essentially traditional programming efforts. If applying those techniques and methods requires essentially the same kind of approach and expertise as other approaches to programming, then they may be very useful, but they are outside the scope of this paper. For example, we exclude the basic application of programming templates or macros; these do not require any special knowledge representation or learning and are standard "tricks of the trade" in software development.

Logic programming languages enable developers to enter in some logical form a set of sentences expressing facts and rules about some problem domain. Examples of such languages include Prolog and Datalog. These can be effective in some domains, but they are still fundamentally programming languages, and thus they are excluded as being outside the scope of this paper.

Probabilistic programming languages unify "general purpose programming with probabilistic modeling; literally, users specify a probabilistic model in its entirety (e.g., by writing code that generates a sample from the joint distribution) and inference follows automatically given the specification" (see http://probabilistic-programming.org). In some cases, probabilistic programming can replace many lines of code with few lines of code (see [Hardesty 2015]). However, it is still programming, so we exclude probabilistic

programming from this paper. Systems designed to generate probabilistic programs may be in scope (e.g., we include [Saad 2019] in this paper).

D. Older Materials

As noted above, the field of automated code generation is vast, and, therefore, we have somewhat arbitrarily limited our survey to relatively recent techniques. We are aware that in so doing we have left out a large body of pertinent previous research, but our assumption is that older promising approaches are likely to have been refined and incorporated in more recent works. That said, it is worthwhile to note some of these materials. The following is an incomplete list:

a. [Bierman 1985] surveys 10 methods for "automatic program construction" and specifically focuses on formal methods for the automatic construction of algorithms from fragmentary information. It divides them into two categories: synthesis from formal specifications and synthesis from examples. It also discusses the possibilities from natural language processing. It notes that all the methods were in active research at that time.

b. [Balzer 1985] summarizes work from 1970 through 1985 on "extended automatic programming," which includes "some means of acquiring the high-level specification to be compiled, some means of determining that it is the intended specification, and some (interactive) means of translating this high-level specification into a lower-level one which can be automatically compiled."

c. [Rich 1992] provides "an overview of current approaches to automatic programming organized around three fundamental questions that must be addressed in the design of any automatic programming system: What does the user see? How does the system work? What does the system know?"

3. Partial Survey of AI Technologies Applicable to Automated Code Generation

In this chapter, we present various approaches for automated source code generation (ASCG) using AI. We also discuss some related work of potential applicability. The approaches were identified via a search of literature published in or since 2010. The results of the literature search led to discussions within IDA, resulting in portions of the commentary reported here. The emphasis is upon relatively recent work; the field of AI is broad and spans the 1950s to the present.

The first section of this chapter is the heart of this paper — a survey of approaches for ASGC using AI. The second section discusses possibly related work that is not focused directly on the specific issue of ASCG, but might provide general ideas on how to approach it. The third section discusses program representation, which is a common theme across a number of different papers and one that appears especially important in solving the ASCG challenge.

The paper identifies several areas that we feel are especially promising and we explain why. However, it should be noted that these are the opinions of a small group of researchers, and that it is very difficult to determine the fruitfulness of a particular research approach ahead of time.

Many (though not all) of the approaches described here use ML. It is important to note that when using an ML approach, there is often a difference between initial training and later performance. A system may perform very poorly during training. Training may or may not occur during performance. A system's true capabilities are only shown during actual performance when it is provided inputs that were not used during previous training.

A. Automated Generation of Source Code Using AI

This section discusses the reviewed AI approaches to ASCG using AI — at least partial generation. We have primarily organized this section by the kind of input used by the AI technology to generate source code and, to some extent, by the kind of output produced. One problem with this organization is that it sometimes separates related technological approaches. However, this organization is much easier to understand than some alternatives, and as noted in [Christakopoulou 2017], "For computers to program computers, we must first address how programming problems will be represented and how performance will be evaluated."

1. Translate from natural language into a program.

    a. [Njonko 2012] argues for a general approach to (1) transform business rules (BRs) expressed in a natural language into Semantics of Business Vocabulary and Business Rules (SBVR), and then (2) translate SBVR into an executable form. They developed a prototype that could generate some executable SQL scripts (e.g., a SQL select statement that could show all records of a given type that failed to meet a specific business rule). The authors believed that SBVR could be useful for this purpose because it is an Object Management Group (OMG) standard specification that "provides a way to capture specifications in natural language (NL) and represent them in formal logic so they can be machine-processed." It is unclear from the paper how difficult it would be to scale up the system to handle "real" business rules, and it would essentially be focused only on business rules when based on SBVR (since that is the focus of SBVR).[7]

    b. [Roychoudhury 2017] describes a semi-automated transformation of the legal NL (English) text to SBVR's Structured English (SE) notation and then to specific executable forms. Again, these papers focus on BRs.

    c. [Desai 2016] takes natural language and produces expressions in a target domain-specific language (DSL). The approach requires training data consisting of pairs of natural language and an equivalent DSL expression. The problem is simplified by focusing on DSLs (instead of general-purpose programming languages) and generating a ranked set of possible programs (instead of a single program).

    d. [Yin 2017] translates natural language statements into a general-purpose programming language, specifically Python. The authors noted that a problem with [Ling 2016] is that "This work treats code generation as a sequence-to-sequence modeling problem, and introduce [sic] methods to generate words from character-level models, and copy variable names from input descriptions. However, unlike most work in semantic parsing, it does not consider the fact that code has to be well-defined programs in the target syntax." To resolve this, [Yin 2017] uses ML techniques to translate natural language into an abstract syntax tree (AST), and then uses deterministic generation tools to translate the AST into

[7] Of course, SBVR is only one example of "structured English." SBVR could be extended, or a different structured language could be used, to accommodate the "primitives" required to capture constructs necessary to support the generation of source code (e.g., by adding the required "structured English" form to express a more general loop). As an example of the potential for extensibility, we note that IDA developed a demonstration (an actual software application) to convert back-and-forth between SBVR and the Object Constraint Language (OCL) used in UML modeling to capture semantics not supported by the graphical language.

final source code. The authors extended the standard RNN decoder to add "additional neural connections which reflect the recursive structure of an AST."

PROMISING. Translating natural language into source code is hard, especially since natural language typically has many ambiguities and background knowledge is important, but some progress has been made. The approach of [Yin 2017] looks especially promising; this approach simplifies the challenge the underlying algorithm must implement and makes it easier for the RNN to discover the recursive structures.

2. Use information from natural language and a structured specification (together) to generate source code.

   a. [Raza 2015] presents a "domain-agnostic program synthesis algorithm and demonstrate[s] its application to an expressive string manipulation language" given a combination of natural language and examples. However, the paper focuses on a very narrow problem domain (manipulation of strings).

   b. [Ling 2016] generates source code from a mixture of "natural language and structured specification." Their approach introduces Latent Predictor Networks (LPNs), a novel neural architecture that computes "the marginal likelihood over latent predictors and generated segments allowing for scalable training." (See also the commentary and extensions of this paper in [Yin 2017].)

   PROMISING. [Raza 2015] is very limited, but [Ling 2016] shows some promise in generating a program from natural language and a structured specification.

3. Use a constrained natural language that looks like natural language but is assigned specific semantics (and thus can be turned into code).

   There is a long history of this approach.[8] However, this approach can be difficult for users; traditional programming languages are themselves constrained languages, so the challenge of using a constrained natural language can be similar to the challenge of writing a program in a traditional programming language.

   a. OMG's *Semantics of Business Vocabulary and Business Rules (SBVR)* [OMG 2017] is a specification that "defines the vocabulary and rules … for documenting the semantics of business vocabularies and business rules for the exchange of business vocabularies and business rules among organizations and between

[8] Although older, it is worth mentioning [Nelson 2006], which discusses *Inform7*. That approach offers a way to program interactive fiction (IF) using a constrained form of English. It is fundamentally rule-based, using rules to map language patterns to underlying models. In this way, it "could be regarded as a descendant of Winograd's program SHRDLU." Many complications ensue because of the complications of English. Interestingly, the constrained language includes ways to define new language processing rules, so it is possible to add new natural language constructs that can be used later.

software tools." The specification's Annex A defines SBVR SE, which defines a rigid form of structured English (SE) and a mapping to the underlying model. This is not an application of AI by itself, but because it has a rich model and a sample notation in a constrained natural language, this is an easier target for translating natural language into something that can be executed. [Bajwa 2010] presents an approach for translating natural languages to SBVR business rules; "a rule based algorithm for robust semantic analysis of English and generat[ion of] SBVR rules" is used.

4. Translate mathematical specifications (or similar notations) into code.

   One term that is sometimes used is *program synthesis*: the task of automatically constructing a program that satisfies a given high-level specification (typically, a non-algorithmic specification). Note that the term *program synthesis* is extremely broad, and we are focusing on a narrower topic. A challenge with these approaches is that they require users to be able to create specifications using mathematics-based notations. At this time, these approaches are only able to synthesize very small programs.

   a. [Rajeev 2013] proposes a unified framework for program synthesis problems that specifically discusses syntax-guided generation.

   b. The annual Syntax-Guided Synthesis Competition (SyGuS-Comp) began in 2014. It is a competition of program synthesis solvers against a variety of benchmarks, where the solvers' goal is to "find a program that meets a correctness specification given as a logical formula." Benchmarks are expressed in the SyGuS input format (SyGuS-IF), a format closely modeled on the SMT-Lib format. More information is available at its website at https://sygus.org/.

   c. [Volkstorf 2015] demonstrates a program synthesis approach that can synthesize PHP programs to meet simple specifications (e.g., to determine if a number is prime or to list the factors of a number).

   d. [Xu 2018] accepts specifications based on real-time process algebra (RTPA) and translates them into MATLAB using an ML approach. "The kernel of the RTPA-MATLAB code generator learns rules from RTPA specifications for both structure and process models in order to automatically generate code in MATLAB… the coding rules elicited from RTPA are represented in the learning engine covering rules of types, primitive operators and relational operators of programs. A finite set of basic rules for code generation is built-in as prior knowledge. The system carries out rule learning under supervision for developing its own programming knowledge base towards automatic code generation." One positive aspect is that the input has rigorous semantics, so the problems of ambiguity in natural language are mostly removed. However, it is not clear that

writing such specifications is any faster or easier than writing code, and it is likely that fewer people can read RTPA specifications than MATLAB.

5. Use structured input-output examples to generate code (this approach is sometimes called "programming by example").

   Much of this work is an application of inductive programming (IP), which "studies the automatic synthesis of computer programs and background knowledge. IP developed from research on inductive program synthesis, now called inductive functional programming (IFP), and from inductive inference techniques using logic, nowadays called inductive logic programming (ILP). IFP addresses the synthesis of recursive functional programs generalized from regularities detected in (traces of) input/output examples using generate-and-test approaches based on evolutionary or systematic search or data-driven analytical approaches… ILP originated from research on induction in a logical framework." [Gulwani 2015]. A discussion about IP is presented in [Gulwani 2015], and the website https://inductive-programming.org/ provides more information about it. However, there are other approaches that have been used as well (as discussed later in this paper).

   Some systems generate programs that attempt to predict future output given past examples of input-output pairs. For our purposes, we include them here.

   a. [Gulwani 2012] presents a programming-by-example approach that allows spreadsheet end users to automate repetitive tasks without knowing how to program. The authors created a domain-specific language (expressive enough to capture several real-world tasks in the domain but also restricted enough to enable efficient learning from examples) and a data structure for representing consistent programs. Their algorithm for synthesizing consistent programs applies "two key procedures: (i) Generate learns the set of all programs, represented using data structure D, that are consistent with a given single example. (ii) Intersect intersects these sets (each corresponding to a different example)." They then ranked the generated programs, "preferring programs that are more general [as inspired by] Occam's razor, which states that a smaller and simpler explanation is usually the correct one." This paper builds on the previous paper [Gulwani 2011], which shows success in automatically generating string manipulation programs in Excel spreadsheets from very few examples. The programs it can generate are small, but they are useful in automating tasks for people who cannot program for themselves. Current versions of Microsoft Excel incorporate this approach for generating a program from data (and then generating new data from the program, though the program is not made visible) in its "flash fill" functionality [Gulwani 2015].

b. [Becker 2013] discusses an experiment using a genetic algorithm to generate code in Brainf-ck. It uses a straightforward approach starting with a population of random genomes, executing them and ranking them by a fitness score, mating the best genomes with mutation to produce a new generation, and repeating until the target score was achieved. Brainf-ck was intended as a joke language, but it is Turing-complete and has only 8 instructions, enabling a very simple approach that generates successful (toy) results. That said, the programs the algorithm was asked to generate were extremely trivial (e.g., "print a short text"), and programs written in the Brainf-ck language are (by design) difficult for humans to understand. It is unclear if this approach could be scaled up to more substantive programs, but it is a remarkably direct approach to the problem.

c. [Kimber 2012] is a PhD dissertation that "presents two novel inductive logic programming (ILP) approaches, based on the notion of a connected theory. A connected theory contains clauses that depend on one another, either directly or via clauses in the background knowledge."

d. Various papers discuss generating source code from input-output examples using gradient descent and differentiable interpreters; examples include [Riedel 2016], [Bunel 2016], and [Gaunt 2016]. However, [Gaunt 2016] also shows that programs created by differentiable interpreters are not as effective as discrete search-based techniques used by the programming languages community.

e. Microsoft's DeepCoder [Balog 2017] improved on previous approaches of source code generation from examples and addressed previous concerns (especially those in [Gaunt 2016]) through the following modifications: "(1) learning how to induce programs; that is, learning how to use a corpus of program induction problems to learn strategies that generalize across problems, and (2) integrating neural network architectures with search-based techniques rather than replace them." DeepCoder received significant press coverage (e.g., [Gershgorn 2017] and [Reynolds 2017]).

f. [So 2018] synthesizes a program that generates a pattern of characters given an input pattern. Their synthesis algorithm combines enumerative search, constraint solving, and program analysis. It is remarkably fast but has only been demonstrated for an extremely narrow problem. That said, using a constraint solver to help generate a program might be useful elsewhere.

g. Synthesis of probabilistic programs. Probabilistic programs can be used to define models (e.g., for analysis, interpretation, or prediction).

    i. [Tong 2016] discusses an approach for building an "expressive probabilistic program from [continuous] time series data when the structure of the model is not given. The intuition behind our method is to find a descriptive

covariance structure of time series data in nonparametric Gaussian process regression."

ii. [Saad 2019] presents a technique for "constructing probabilistic programs for data analysis, interpretation, and prediction" and demonstrates its use for obtaining "predictions of new time series data and new multivariate data records."

PROMISING. Determining what a program should be from a set of input-output pairs is hard, even for humans, because there are a very large number of programs that could generate them. More broadly, generalization from any set of examples is likely to be a difficult task if the objective is extrapolation rather than interpolation (i.e., expecting to be able to reason in a situation outside the boundaries of the dataset). Even interpolation can be difficult, if by "small" one means "sparse" (lots of blank space with no examples in the problem domain). IP synthesizes computer programs by defining and exploiting background knowledge; this has had a number of successes at small scales (e.g., generating spreadsheet string transformations from examples as shown in [Gulwani 2011], [Gulwani 2012], and [Gulwani, 2015] and implemented in Microsoft Excel as "flash fill"). Synthesis of programs to model time-continuous values (e.g., to predict future values) has significant promise. Microsoft's DeepCoder [Balog 2017] uses approaches to learn what programs should look like, and that also appears to be a promising approach to aid in program generation from a set of examples.

6. Generate source code from a text query and predefined cases using case-based reasoning.

   a. [Danilchenko 2012] takes a user specification in text, specifically a medical database query, and generates equivalent Java code. It uses a combination of Case-Based Reasoning, Routine Design and Template-Based Programming. The Automated Coder using Artificial Intelligence (ACAI) system requires cases to be defined using XML. A stated advantage is that users do not need to learn SQL.[9] It is not clear how promising this approach is. Currently "ACAI only solves database-type problems" focusing on the medical database domain (though the authors argue that it could be expanded beyond this). The authors have only demonstrated a modest ability to combine plans to produce relatively simple database queries, and it requires cases to be defined using XML. In short, this approach requires a lot of basic upfront work to perform trivial tasks, and it is not

[9]Someone still has to learn XML and encode cases using XML instead of requiring the end-user to learn SQL. It could be debated what is harder.

clear how easily the approach could be expanded to more real-world circumstances.

7. Use generate-and-validate patch generation systems.

   Generate-and-validate patch systems "start with a program and a suite of test cases, at least one of which exposes a defect in the program. The systems then generate a space of candidate patches and search this space to find plausible patches that produce correct outputs for all test cases in the test suite" [Long 2016].[10]

   a. Earlier generate-and-validate patch generation systems include those described in [Samimi 2012], [Nguyen 2013], [Samanta 2014], and [Kneuss 2015]. However, [Long 2016] claims that they are typically limited to only work on small programs with hundreds of lines of code.

   b. Prophet is a novel patch generation system from MIT that uses a learned model of correct code to rank the patches in its search space, with the goal of obtaining a correct patch as the first patch (or one of the first few patches) to validate. It is able to handle larger applications from tens of thousands to a million lines of code [Long 2016].

      PROMISING. Patch generation to repair existing code is a narrower problem, but patch generation systems are extremely useful, and MIT's Prophet [Long 2016] has been demonstrated on larger codebases.

8. Accept natural language and generate UML diagrams (such as class diagrams) that can then be turned into code.

   One challenge is that UML diagrams focus on the overall architecture of a software system, so the diagrams provide guidance for only some of the necessary code to be created.

   a. Requirement Analysis to Provide Instant Diagrams (RAPID) is "a desktop tool to assist humans to analyze textual requirements and extract UML diagrams [specifically class diagrams]. The evaluation of the RAPID system is in process and will be conducted through two forms of evaluation, experimental and expert evaluation." [More 2012]. However, the approach is not flexible and is limited in what it can produce.

   b. [Sharma 2015] translates natural language into UML class diagrams. To improve the results, they "transform the requirements statements to an intermediary frame-

[10] A more in-depth overview and discussion can be found in "*The White-Hat Hacking Machine: Meet Mayhem, winner of the DARPA contest to find and repair software vulnerabilities*", IEEE Spectrum, Year: 2019, Volume 56, Issue 02, pp. 30–35. It should also be noted that the systems that resulted were operating upon very large code bases (e.g., operating systems).

based structured representation using dependency analysis of requirements statements and the Grammatical Knowledge Patterns. The knowledge stored in the frame-based structured representation is used to derive class diagrams using [a] rule-based algorithm."

c. [Gulia 2016] takes natural language text and generates UML diagrams (specifically activity diagrams and sequence diagrams). This approach is more symbolic, using the Stanford part-of-speech (POS) tagger and Stanford parser to parse the sentences. Simple rules are then used to transform this information into UML diagrams.

d. [Narawita 2016] generates UML use case diagrams and class diagrams from natural language. It uses a simple natural language processing library coupled with basic ML. It was tested "with more than twenty (20) scenarios and it has an accuracy level of around 70%." Note that such a system would typically need to have a higher reliability to be useful. This system was developed as an undergraduate project, so while this particular system is not very capable, it demonstrates that natural language processing libraries are easy enough to use so that an undergraduate project can use one (though not necessarily with strong results).

PROMISING. It is possible to convert natural language to UML diagrams and then use them to generate some code (see [More 2012], [Sharma 2015], and [Gulia 2016]). However, this is very limited – most code would not be generated by this technique, and the technique requires a programming-level understanding of what to state in natural language. It is not clear how fruitful this approach can be for the general case of ASCG.

9. Generate source code given a label (for example, a set of API calls or types) carrying a small amount of information about the code that is desired and a corpus of labeled programs.

   a. BAYOU generates Java code given labels and a corpus — training not on code but on program sketches — and concretizing them. BAYOU is described in [Murali 2018]. The work was funded by Google and the US military. BAYOU is publicly available at https://github.com/capergroup/bayou under the Apache 2.0 open source software license. It is not clear how promising this is; the approach is interesting, but the need for labels is a concern.

10. Allow users to provide an image (such as a hand drawing or a screenshot) describing a UI and then translate that image into code that implements the UI.[11] The resulting code or data might be edited later using a UI editor.

    a. Beltramelli of Ulzard Technologies discusses pix2code in [Beltramelli 2017]. This generates code from a graphical user interface (GUI) screenshot using deep learning methods. They managed "to automatically generate code from a single input image with over 77% of accuracy for three different platforms (i.e. iOS, Android and web-based technologies)."[12]

    b. [Wilkins 2018] discusses Airbnb's sketch2Code system, which is "able to scan the mockups made by the designers and translate them into code." (Summary by [Cheng 2018]). In short, users were able to hand-draw representations of a UI (including indicators for different kinds of UI components), and the system was able to generate code that implemented the UI. The goal was to rapidly test designs. Their prototype supported about a dozen different kinds of hand-drawn components. The authors were pleasantly surprised by the results, and they believe it has potential.

       PROMISING. Converting hand-drawn UI sketches into code is somewhat promising as it can potentially reduce the human effort required to specify the UI. Both [Beltramelli 2017] and [Wilkins 2018] have demonstrated some useful prototypes, although they are expressly limited in scope.

11. Generate test cases using operators, an initial state, and a goal state.

    a. [Memon 2001] presents Planning Assisted Tester for grapHical user interface Systems (PATHS), which automatically generates "test cases for [Graphical User Interfaces [GUIs]) that [exploit] automated planning, a well-known technique in artificial intelligence." Given a set of operators, an initial state, and a goal state, a planner produces a sequence of operators that transforms the initial state into the goal state. "This approach is designed to create test cases, not application code." However, test cases are often created by writing code, so this can be viewed as a way to write code.

---

[11] A more detailed analysis is required to assess the value added from such an approach and what is already supported by UI editing tools like Visual Studio (VS). Such tools allow users to create and edit UIs using a GUI. The approaches discussed here support source code generation from screenshots or hand-drawn sketches, rather requiring something be specifically "drawn" via UI editing tools. It could be argued that requiring users to use a UI editing tool is not difficult, and thus mechanisms to generate UIs from other input sources are not an adequate improvement to be worthwhile.

[12] 77% accuracy is almost certainly unacceptable for a software development environment intended to produce apps that will be used by soldiers deployed in theaters of operation. It is arguable that accuracies in the range of 99%+ is what is needed.

12. Detect defects (bugs) in code.

    This approach is not focused on source code generation; instead, it focuses on detecting bugs[13] in existing code. However, it seems worth mentioning because using AI to speed development of quality code seems relevant to the overall goal. This approach could theoretically be used in tandem with code generated by other AI-based approaches to increase the quality of the final result.

    a. [Chappelly 2017] used ML to find defects. In their approach, "While on the surface the initial results were encouraging, further investigation suggests that the machine learning techniques we used are not suitable replacements for static program analysis tools due to low precision of the results. This could be due to a variety of reasons including not using domain knowledge such as the semantics of the programming language and lack of suitable data used in the training process." We suspect that the lack of suitable data was a key reason for this problem; they were using an ML approach based on statistical methods, but their data sets were small because they were from hand-created code (80% artificial samples of small correct and incorrect code, 20% fragments of real code).

    b. DeepBugs [Pradel 2018] "reasons about names based on a semantic representation." It learns likely incorrect code examples by using simple code transformations to create artificially-seed defects. This approach "revealed 102 programming mistakes (with 68% true positive rate) in real-world code." Some additional information is in [Pradel 2017]. Its implementation is available at https://github.com/michaelpradel/DeepBugs. This seems to have been more successful; we suspect that is because they could have a far larger training dataset through artificially seeded defects.

       PROMISING. The literature surveyed here is about detecting software defects, instead of actually writing code; however, simplifying the problem can make it more immediately tractable. DeepBugs [Pradel 2018] suggests that creating larger datasets[14] by using techniques such as artificially seeded defects may be key to progress. That said, there are already many static analysis tools including compiler warning flags, source code quality analyzers ("style checkers"), and source code weakness analyzers that successfully find defects without these approaches [Wheeler 2016]. These new approaches do not need to completely replace current approaches, but they do need to find enough defects without too

[13] See https://www.kestrel.edu/home/projects/apt/.

[14] This is consistent with the finding from the competitors in the DARPA challenge — all of the systems were supercomputer-class systems.

many false positives compared to existing technologies. It remains to be seen if these alternative approaches will become robust enough to use them as well.

13. Use a different internal model designed for representing source code when processing source code.

    This is not an approach that can be used in isolation; instead, it is a potential underlying technological enhancement to many of the approaches above.

    a. [Hellendoorn 2017] argues that a different language modeling approach can be more effective when processing source code, because source code has special properties (e.g., new identifiers proliferate and there are deeply nested scopes). They present an adapted N-gram model for source code and show that it has some improved measures (e.g., a higher probability to predict a "next token" in program).

14. Provide a program to score whether or not the generated program meets the objective, and use the scoring program source code (as well as possibly executing it).

    This approach requires writing a program in the first place, which may seem unsatisfactory as that requires development expertise. However, it may be easier to write a program to verify that an answer is correct than to write the program that produces correct answers. The approach is interesting, but the requirement to write a program makes it harder to recommend.[15]

    a. [Christakopoulou 2017] accepts the source code of a program that can verify if another program meets a specification and uses the output of the verification to guide the generation of the program that meets the specification.[16] They call this a "glass box" approach because they use the information from the verifier's source code.

15. Generate "random" code to see if it will run.

    This is an "obvious" way to use some ML techniques, especially as there have been systems trained to "write" works by past authors such as Shakespeare. The fundamental problem with this approach is that usually just generating a running program is not the goal; you can go to sites like GitHub or GitLab and download large sets of runnable programs. In most cases people need programs that perform specific tasks, and this approach fails to do this. This approach could be useful in

[15] One approach to strong forms of verification is theorem proving, which is difficult but has been used quite successfully in some domains, such as Macsyma's work (1970s and all the symbolic math systems that followed), as well as John Gabriel's work (early 1980s) using theorem provers to verify safety/reliability of nuclear power systems. Formal methods (applying mathematical techniques to specifications and models) is its own domain and is beyond what we cover in this paper.

[16] This may work best, or perhaps only, with constrained source code languages.

highly specialized cases (e.g., as a way to provide test data for compilers), but those are narrow uses. We further note that even though pure random program generation is probably of little utility, randomness guided by methods such as genetic programming can have a great deal of utility. For example, this approach has been successfully used to program neuromorphic computing technologies [Schuman 2015].

a. [Priya 2017] discusses automatically generating essentially random code for a variety of programming languages using character-based RNNs. They managed to demonstrate its ability to create code for a variety of problems (which is impressive), but they are all small, and the requirement to write a program in the first place is a limiting factor.

16. Interactively ask the user questions when a rule-based system needs to know what direction to take.

    a. [Imam 2014] describes code generation with an expert code generator using rule-based and frames knowledge representation techniques (ECG-RF). Predefined frames of fixed structures are filled with code chunks from a knowledge base by inferencing system, which is guided by the user. The user is asked questions, and the system responds based on those answers. One example the authors give generates assembly code for a DOS driver. This approach is easy to understand and clearly can work, but rule-based systems can be challenging to build as the rules get more complex, and the paper by itself doesn't suggest how to deal with that.

17. Support code completion.

    Instead of trying to write the whole program, use techniques to recommend specific code fragments while a human is writing source code.

    a. [Li 2018] describes a code-completion mechanism that manages to do well even without type information (e.g., for a language like Python). The discussion may provide insights for other approaches as well:

        i. The authors note that "(standard) neural language models such as Recurrent Neural Networks (RNNs) can capture sequential distributions and deep semantics (but) are limited by the so-called hidden state bottleneck: all the information about current sequence is compressed into a fixed-size vector. The limitation makes it hard for RNNs to deal with long-range dependencies, which are common in program source code…" Their solution is to use abstract syntax tree (AST) representations instead of simple text and use a tailored "attention mechanism" so the system can learn to retrieve and make use of relevant previous hidden states. As they note, "Representing programs

as ASTs rather than plain text enables us to predict the structure of the program, i.e., type of each AST node."

ii. Words (such as variable names) tend to repeat locally, even if they are rare globally. They create "pointer mixture" to try to predict when to use the globally common vocabulary and when to use local words.

PROMISING. Code completion appears promising because the problem appears to be much easier (only small fragments need to be generated, not an entire program) and the impact of inaccuracy is greatly reduced (the user is a software developer who can confirm if the suggested code is correct and can change it if it is incorrect). It also fits easily into existing integrated development environments (IDEs), which often already have some mechanisms for interactive recommendations. In addition, [Li 2018] provides additional evidence that focusing on underlying abstract syntax trees (ASTs) may be a better representation for programs for this purpose. Extending learning with an "attention mechanism" may also be valuable in helping learning systems to learn from long-range interconnections.

18. Aid in generalization and refinement of code templates to help developers search or transform source code.

    a. [Molderez 2016] describes an approach that starts with an existing system (EKEKO/X) that enables developers to develop code templates for searching or transforming source code. Their approach uses genetic algorithms and fitness functions to automatically generalize and refine a template group (so it matches only what is desired). This helps existing software developers develop software, and it depends on knowledgeable developers, as they will need to work at a very abstract level. It is unclear how much effort would be saved in practice (no such experimental data is provided in the paper), which is important to know about such a different approach to supporting developers.

19. Other related surveys.

    a. [Allamanis 2018] is an extensive survey of papers relating to ML and ASCG, including some directly relevant papers and discussions about representations that could be helpful. Interested readers should also investigate this survey. It is worth noting that the resulting programs (or program fragments) need not be perfect. It would be possible for systems to create "draft" code to be reviewed by humans.[17]

[17] It is also possible for humans to create draft code to be reviewed and improved by computers. Indeed, one of the approaches we listed earlier specifically looks for defects. Tools that report likely defects (and possibly suggested fixes) to human software developers can be very helpful as long as the false positive rate for reports is not too high. Many software developers use such tools today. However, if the tool's

B. Possibly Related Work

The following do not *directly* apply to ASCG, but might hint at solutions:

1. There are various efforts to generate natural language from source code (i.e., the opposite direction of ASCG). For a survey, see [Neubig 2016].
2. Various tools have been developed to create mathematical proofs using AI techniques (e.g., [Whalen 2016]). Mathematical proofs and programs do share some features, so these approaches might be relevant.[18]
3. Solutions to well-known AI problems have become far more capable over the last few years. Some of those solutions may have lessons for generating source code as well:
    a. Google's AlphaZero algorithm can achieve superhuman performance in many challenging games, specifically Chess, Shogi, and Go [Silver 2018]. This was achieved using no domain knowledge except the game rules.[19]
    b. [Jaderberg 2019] most recently demonstrated that AI techniques, such as RNN, can be used in other game categories, such as Capture the Flag, where two multiplayer teams compete in capturing the flags of the opposing team.
    c. Stacked Generative Adversarial Networks (StackGAN) can generate 256x256 photo-realistic images conditioned on text descriptions [Zhang 2017].
    d. WaveNet is a deep neural network for generating raw audio waveforms. It can generate audio from text that is more natural sounding than many previous systems [Oord 2016].
4. A large amount of effort and creativity is ongoing to develop new and/or possibly improved techniques in AI.
    a. RNNs are neural networks that support training of sequences, which is relevant for tasks such as natural language translation and generating source code. The RNN approach has been very effective for many problems, as noted in [Karpathy 2015].
    b. Generative adversarial networks (GANs) and RN are extremely general techniques for improving capabilities over time.

---

results would be directly executed without first being reviewed by a software developer, then, in many cases, the tool would need to be very nearly perfect. This points to one of the serious limitations in the current state-of-the-art for many AI technologies: Accuracy is not always their strong suit.

[18] There is a long and rich history of computer-generated proofs dating at least back to the 1970s.

[19] Technically, the game's rules were utilized to generate example simulations of the games, and those were then used to train the systems.

c. [Chen 2018] introduces a new family of deep neural network models (instead of specifying a discrete sequence of hidden layers, the derivative of the hidden state is parameterized using a neural network).

5. There are many libraries / frameworks / tools to help develop AI/ML systems. In many cases, these are open source software (OSS) and are typically free to download and improve:

   a. TensorFlow (https://www.tensorflow.org/): TensorFlow is an open source software library for high performance numerical computation (including those for ML). For example, TensorFlow was used on the world's most powerful supercomputer (Summit at DOE's Oak Ridge National Laboratory) as part of a climate change research experiment and achieved exaflop-level performance (one exaflop is 10^{18} floating point operations per second). Google employees helped the project adopt TensorFlow to Summit's scale and expect to use the experience they gained elsewhere [Simonite 2019]. This is an interesting example where government and industry have collaborated (including via the use of OSS).

   b. Keras (https://keras.io/): Keras is a high-level neural network API, written in Python and capable of running on top of TensorFlow, the Microsoft Cognitive Toolkit (CNTK), or Theano.

   c. Scikit-learn (https://scikit-learn.org/): Scikit-learn is an ML system implemented in Python.

   d. Dlib (http://dlib.net/): Dlib is a C++ toolkit containing ML algorithms and tools for creating complex software in C++ to solve real world problems.

   e. For completeness we also include the R programming language which contains a large number of ML libraries (https://www.r-project.org/)

6. There are many existing databases and pre-trained models. ML algorithms generally require large datasets for training. Transfer learning methods can utilize pre-trained models for related problems to train models for new applications. The databases may be specifically developed for ML or may simply be datasets that are available. In the case of software, the large amount of software available as OSS through GitHub, GitLab, and other repositories may be useful as part of a training dataset.

C. Representations of Programs

Several approaches emphasize creating a higher-level abstraction for source code rather than trying to directly generate text. For example, several use ASTs [Yin 2017] [Li 2018].

We think using representations that easily model these higher-level structures may make it easier to use well-known techniques such as RNN models, as well as making it easier to extend those techniques to deal with the underlying structure of source code. Such higher-level models should make it easier for learning systems to identify patterns and create syntactically correct code (e.g., ensuring that block ending marks match block beginning marks).

4. Conclusions

ASCG using AI is not a solved problem. Many approaches have only worked in laboratory settings and/or only on small-scale problems. If the need is to develop industrial-scale software today, then more conventional approaches are indicated in almost all cases.

That said, we identified eight promising research efforts, but found the following six most warrant further research funding:

1. Determining what a program should be from a small set of input/output pairs is hard, even for humans, because a large number of programs could generate that data. However, it has been done. IP approaches have successfully developed small programs, such as inferring string manipulation programs in spreadsheets (an approach that is implemented in Microsoft Excel's "flash fill" function). Microsoft's DeepCoder [Balog 2017] uses IP approaches to learn what programs should look like, which can aid in program generation.

2. Translating natural language into source code is hard, especially since typically natural language has many ambiguities, and background knowledge is important. However, the approach in [Yin 2017] looks especially promising — it simplifies the challenge the underlying algorithm must implement and makes it easier for an RNN to discover the recursive structures.

3. Patch generation to repair existing code is a narrower problem but extremely useful, and MIT's Prophet [Long 2016] has been demonstrated on larger codebases.[20]

4. It is possible to convert natural language to UML class diagrams and then use those diagrams to automatically generate source code stubs and the general program structure (see [More 2012], [Sharma 2015], and [Gulia 2016]). This approach all by itself is somewhat limited — the source code for the methods and functions that make up the body of the code stubs would not be generated by this technique, and a programming-level understanding of what to state in natural language is required. However, this approach could be fruitful in combination with some of the other ASCG techniques discussed in this paper.

[20] See https://www.kestrel.edu/home/projects/apt/.

5. Converting UI sketches into code is promising — both [Beltramelli 2017] and [Wilkins 2018] demonstrate some useful prototypes, although they are expressly limited in scope.

6. Creating a higher-level abstraction for source code, instead of trying to directly generate text. For example, several of the approaches discussed in this paper use ASTs (see [Yin 2017] [Li 2018]). Such higher-level models should make it easier for learning systems to identify patterns and create syntactically correct code (e.g., ensuring that block ending marks match block beginning marks). Using representations that easily model these higher-level structures should make it easier to use well-known techniques such as RNNs and could possibly make it easier to extend those techniques to deal with the underlying structure of source code.

It should be kept in mind that these results are in some sense dependent on how one defines AI. We have expressly excluded (1) general-purpose programming language compilers developed using traditional techniques, (2) MDE / MDA, and (3) CASE tools, IDEs, and traditional code generators. All of these are in routine use today in developing software, but they are not typically included in the term AI. We have also assumed, for this paper, that automatically generating source code is the goal; it is possible to develop systems where there is no conventional source code (e.g., learn-by-example techniques in robotics), but we have excluded that as out-of-scope.

Given the ever-growing importance of software in all spheres of modern life, we expect that research in this area will accelerate in the future and that more effective methods will be developed and eventually commoditized.

Annotated References

We have annotated many of these references (e.g., by including their abstracts and/or extracts) to make it easier to understand what the references are about. URLs are provided to ease access; some URLs may no longer be valid.

[Allamanis 2018] Allamanis, Miltiadis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton, "A Survey of Machine Learning for Big Code and Naturalness," 2018, *ACM Computing Surveys*, Vol. 51, No. 4, Article 81. Publication date: July 2018.

> Abstract: "Research at the intersection of machine learning, programming languages, and software engineering has recently taken important steps in proposing learnable probabilistic models of source code that exploit the abundance of patterns of code. In this article, we survey this work. We contrast programming languages against natural languages and discuss how these similarities and differences drive the design of probabilistic models. We present a taxonomy based on the underlying design principles of each model and use it to navigate the literature. Then, we review how researchers have adapted these models to application areas and discuss cross-cutting and application-specific challenges and opportunities."

[Almeida 2011] Almeida, Frade, Pino, and Melo de Sousa, 2011, *Rigorous Software Development: An introduction to Program Verification*, http://www.springer.com/cda/content/document/cda_downloaddocument/9780857290175-c2.pdf?SGWID=0-0-45-1053837-p174029011

[Bajwa 2010] Bajwa, Imran S., Mark G. Lee, and Behzad Bordbar. "SBVR Business Rules Generation from Natural Language Specification." Association for the Advancement of Artificial Intelligence (AAAI). 2010. https://www.researchgate.net/publication/210346459_SBVR_Business_Rules_Generation_from_Natural_Language_Specification/download

> Abstract: "In this paper, we present a novel approach of translating natural languages specification to SBVR business rules. The business rules constraint business structure or control behaviour of a business process. In modern business modelling, one of the important phases is writing business rules. Typically, a business rule analyst has to manually write hundreds of business rules in a natural language (NL) and then manually translate NL specification of all the rules in a particular rule language such as SBVR, or OCL, as required. However, the manual

translation of NL rule specification to formal representation as SBVR rule is not only difficult, complex and time consuming but also can result in erroneous business rules. In this paper, we propose an automated approach that automatically translates the NL (such as English) specification of business rules to SBVR (Semantic Business Vocabulary and Rules) rules. The major challenge in NL to SBVR translation was complex semantic analysis of English language. We have used a rule based algorithm for robust semantic analysis of English and generate SBVR rules. Automated generation of SBVR based Business rules can help in improved and efficient constrained business aspects in a typical business modelling."

[Balog 2017] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. "Deepcoder: Learning to Write Programs." ICLR 2017. https://openreview.net/pdf?id=ByldLrqlx

Abstract: "We develop a first line of attack for solving programming competition-style problems from input-output examples using deep learning. The approach is to train a neural network to predict properties of the program that generated the outputs from the inputs. We use the neural network's predictions to augment search techniques from the programming languages community, including enumerative search and an SMT-based solver. Empirically, we show that our approach leads to an order of magnitude speedup over the strong non-augmented baselines and a Recurrent Neural Network approach, and that we are able to solve problems of difficulty comparable to the simplest problems on programming competition websites."

Introduction says: "Recently, there has been much interest in program-like neural network models (Graves et al., 2014; ...) but none of these can write programs; that is, they do not generate human-readable source code. Only very recently, Riedel et al. (2016); Bunel et al. (2016); Gaunt et al. (2016) explored the use of gradient descent to induce source code from input-output examples via differentiable interpreters, and Ling et al. (2016) explored the generation of source code from unstructured text descriptions. However, Gaunt et al. (2016) showed that differentiable interpreter based program induction is inferior to discrete search-based techniques used by the programming languages community. We are then left with the question of how to make progress on program induction using machine learning techniques. In this work, we propose two main ideas: (1) learn to induce programs; that is, use a corpus of program induction problems to learn strategies that generalize across problems, and (2) integrate neural network architectures with search-based techniques rather than replace them. In more detail, we can contrast our approach to existing work on differentiable interpreters. In differentiable interpreters, the idea is to define a differentiable mapping from source code and

inputs to outputs. After observing inputs and outputs, gradient descent can be used to search for a program that matches the input-output examples. This approach leverages gradient-based optimization, which has proven powerful for training neural networks, but each synthesis problem is still solved independently—solving many synthesis problems does not help to solve the next problem. We argue that machine learning can provide significant value towards solving Inductive Program Synthesis (IPS) by re-casting the problem as a big data problem."

Most of the authors are from Microsoft Research.

[Balzer 1985] Balzer, Robert, "A 15 Year Perspective on Automatic Programming," *IEEE Transactions on Software Engineering*, November 1985, pp. 1257–1268, Vol. 11, DOI 10.1109/TSE.1985.231877, https://www.computer.org/csdl/journal/ts/1985/11/01701945/13rRUxZzAj1

Abstract: "Automatic programming consists not only of an automatic compiler, but also some means of acquiring the high-level specification to be compiled, some means of determining that it is the intended specification, and some (interactive) means of translating this high-level specification into a lower-level one which can be automatically compiled."

[Becker 2013] Becker, Kory. "Using Artificial Intelligence to Write Self-Modifying/Improving Programs." 2013-01-27 modified 2013-03-04. http://www.primaryobjects.com/2013/01/27/using-artificial-intelligence-to-write-self-modifying-improving-programs/

Code available at https://github.com/primaryobjects/AI-Programmer

Extract: "This article describes an experiment to produce an AI program, capable of developing its own programs, using a genetic algorithm implementation with self-modifying and self-improving code… This experiment was a proof-of-concept that an AI program could develop its own computer programs that perform a specific task. In that regard, it was a success."

This experiment uses a genetic algorithm to generate code in Brainf-ck. Brainf-ck was intended as a joke language, but it is Turing-complete and only has 8 instructions, enabling a very simple approach that leads to successful (toy) results. That said, the programs the algorithm was asked to generate were extremely simple (e.g., "print a short text"), and programs written in the Brainf-ck language are (by design) difficult for humans to understand. It is unclear if this approach can be scaled to substantial programs, but it is a remarkably direct approach to the problem.

[Beltramelli 2017] Beltramelli, T. of Ulzard Technologies. "pix2code: Generating Code from a Graphical User Interface Screenshot." 2017-09-17. https://arxiv.org/pdf/1705.07962v2.pdf

Abstract: “Transforming a graphical user interface screenshot created by a designer into computer code is a typical task conducted by a developer in order to build customized software, websites, and mobile applications. In this paper, we show that deep learning methods can be leveraged to train a model end-to-end to automatically generate code from a single input image with over 77% of accuracy for three different platforms (i.e. iOS, Android and web-based technologies).”

Quotation: “The process of implementing client-side software based on a Graphical User Interface (GUI) mockup created by a designer is the responsibility of developers. Implementing GUI code is, however, time-consuming and prevent developers from dedicating the majority of their time implementing the actual functionality and logic of the software they are building.”

[Biermann 1985] Biermann, A. W. (1985). “Automatic Programming: A Tutorial on Formal Methodologies.” *Journal of Symbolic Computation*, 1, pp. 119–142. https://www.sciencedirect.com/science/article/pii/S0747717185800109

Abstract: “Ten methodologies for automatic program construction are presented, discussed and compared. Some of the techniques generate code from formal input-output specifications while others work from examples of the target behaviour or from natural language input.”

Extract: “The formal methodologies - have been separated into two categories, synthesis from formal specifications and synthesis from examples. In the former case, it is assumed a specification is given for the target program with adequate domain information so that the target program can be derived in a series of logical steps. In the latter case, behavioural examples are given for the desired program, and it is inferred by a series of generalization steps. After completing the coverage of these formal methodologies, a short section mentions some work on the generation of programs from natural language input using artificial intelligence knowledge based systems. The various synthesis methodologies will be described by illustrating their operation on a single programming problem.”

Interestingly, the paper uses Common Lisp concepts (such as list, atom, nil (as equivalent to the empty list), car, cdr, and cons) but uses a more mathematical (non-Lisp) syntax for operations (e.g., car(x)).

[Black 2016] Black, Paul E., Lee Badger, Barbara Guttman, and Elizabeth Fong, 2016, *Dramatically Reducing Software Vulnerabilities: Report to the White House Office of Science and Technology Policy*, NISTIR 8151, http://nvlpubs.nist.gov/nistpubs/ir/2016/NIST.IR.8151.pdf

[Bunel 2016] Bunel, Rudy R, Alban Desmaison, Pawan K Mudigonda, Pushmeet Kohli, and Philip Torr. “Adaptive Neural Compilation.” In *Proceedings of the 29th Conference on Advances in Neural Information Processing Systems (NIPS)*, 2016.

[Chappelly 2017] Chappelly, Timothy, Cristina Cifuentes, Padmanabhan Krishnan, and Shlomo Gevay. “Machine Learning for Finding Bugs: An Initial Report.” *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*. 2017. https://ieeexplore.ieee.org/document/7882012

> Abstract: “Static program analysis is a technique to analyse code without executing it, and can be used to find bugs in source code. Many open source and commercial tools have been developed in this space over the past 20 years. Scalability and precision are of importance for the deployment of static code analysis tools - numerous false positives and slow runtime both make the tool hard to be used by development, where integration into a nightly build is the standard goal. This requires one to identify a suitable abstraction for the static analysis which is typically a manual process and can be expensive. In this paper we report our findings on using machine learning techniques to detect defects in C programs. We use three off-the-shelf machine learning techniques and use a large corpus of programs available for use in both the training and evaluation of the results. We compare the results produced by the machine learning technique against the Parfait static program analysis tool used internally at Oracle by thousands of developers. While on the surface the initial results were encouraging, further investigation suggests that the machine learning techniques we used are not suitable replacements for static program analysis tools due to low precision of the results. This could be due to a variety of reasons including not using domain knowledge such as the semantics of the programming language and lack of suitable data used in the training process.”

[Chen 2018] Chen, Ricky T. Q., Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. *Neural Ordinary Differential Equations.*

> Abstract: “We introduce a new family of deep neural network models. Instead of specifying a discrete sequence of hidden layers, we parameterize the derivative of the hidden state using a neural network. The output of the network is computed using a blackbox differential equation solver. These continuous-depth models have constant memory cost, adapt their evaluation strategy to each input, and can explicitly trade numerical precision for speed. We demonstrate these properties in continuous-depth residual networks and continuous-time latent variable models. We also construct continuous normalizing flows, a generative model that can train by maximum likelihood, without partitioning or ordering the data dimensions. For training, we show how to scalably backpropagate through any ODE solver, without

access to its internal operations. This allows end-to-end training of ODEs within larger models."

[Cheng 2018] Cheng, John. *Artificial Intelligence and Auto-generation of Code*. 2018-05-13. Bachelor's Thesis, Degree Program in Business Information Technology, Haaga-Helia University of Applied Sciences.

Abstract: "… The idea is to feed the machine with a design mockup and based on that, it will be able to recognize and return a code based on the different components and layout found. By reading this thesis, neophytes will be able to understand the different steps needed to explore a solution with deep explanation of what machine learning is and how to use it for visual recognition. Furthermore, as machines are extremely good at reproducing, a way to facilitate teams transitions in projects is also detailed. Finally, a solution is provided where a design will be fed to the Artificial Intelligence and an HTML code is received. Being a subject relatively new for the time of the thesis, the material and help to implement the final solution are limited. That is why, even incomplete, it should be considered as a boilerplate to continue working on to further improve it."

This student attempted to do something extremely ambitious as a Bachelor's project, and it did not work out. The work here might lead to further work, but the results within the time available were not successful. "After going through the implementation part, the solution is, unfortunately, not a working one. … [I] would say that my proof of concept is not complete but can be used as a boilerplate to explore new ways, new leads." The document also points to [Wilkins 2018] and [Beltramelli 2017].

[Christakopoulou 2017] Christakopoulou, Konstantina (University of Minnesota) and Adam Tauman Kalai (Microsoft Research), "Glass-Box Program Synthesis: A Machine Learning Approach," arXiv:1709.08669v1 [cs.LG] 25 Sep 2017

Abstract: "Recently proposed models which learn to write computer programs from data use either input/output examples or rich execution traces. Instead, we argue that a novel alternative is to use a glass-box loss function, given as a program itself that can be directly inspected. Glass-box optimization covers a wide range of problems, from computing the greatest common divisor of two integers, to learning-to-learn problems. In this paper, we present an intelligent search system which learns, given the partial program and the glass-box problem, the probabilities over the space of programs. We empirically demonstrate that our informed search procedure leads to significant improvements compared to brute-force program search, both in terms of accuracy and time. For our experiments we use rich context free grammars inspired by number theory, text processing, and algebra. Our results show that (i) performing 4 rounds of our framework typically solves about 70% of the target problems, (ii)

our framework can improve itself even in domain agnostic scenarios, and (iii) it can solve problems that would be otherwise too slow to solve with brute-force search."

[Danilchenko 2011] Danilchenko, Yuri B., *Automatic Code Generation Using Artificial Intelligence*, 2011-07-11. Committee chair Richard Fox, Committee members Wei Hao, Jeff Warn, and Frank Braun, Director Maureen Doyle, Thesis for Master of Science in Computer Science at Northern Kentucky University.

Abstract: "Automatic Code Generation (ACG) allows software engineers to create more concise, maintainable, and reusable solutions, ultimately improving their productivity. Despite the significant benefits and the profound economic impact of ACG in the software development field, it still often requires substantial input and interaction from humans. The presented Automatic Coder using Artificial Intelligence (ACAI) system uses a novel approach to solve fully automated code generation in routine programming domains. Given high level user goals and preferences, a library of abstract programs, and a library of generic code components, ACAI uses a combination of Case-Based Reasoning, Routine Design, and Template-Based Programming approaches to generate complete Java programs that satisfy user requirements."

[Danilchenko 2012] Danilchenko, Yuri, and Richard Fox (Department of Computer Science, Northern Kentucky University). *Automated Code Generation Using Case-Based Reasoning, Routine Design and Template-Based Programming*.

Abstract: "Automated code generation is the process whereby a computer program takes user specifications in some form and produces a program as output. Automated code generation can be the process undertaken by a compiler, which generates an executable program from a source program, but it also applies to the situation where the input is a task described at some level of abstraction and the output is a program that can perform that task. Several different approaches have been utilized to varying degrees of success to automate code generation, including Case-Based Reasoning, formal methods and evolutionary algorithms. In this paper, a system is introduced which combines Case-Based Reasoning, Routine Design and Template-Based Programming to generate programs that handle straight-forward database operations. This paper presents the approach taken and offers some brief examples."

Extracts and comments: This paper presents the "Automated Coder using Artificial Intelligence (ACAI) system." ACAI accepts the goal (text describing the query or queries to be answered) and specifications for how to achieve the goal ("e.g. computational complexity, memory and disk usage, form of input, form of output"). A case must be retrieved from a library of cases (plans described using XML); these must be created by developers. The system generates Java source code. Currently,

"ACAI only solves database-type problems" focusing on the medical database domain, though the authors argue that it can be expanded beyond this. It has only demonstrated a modest ability to combine plans to produce relatively simple database queries. A key advantage is that users do not need to learn SQL.

This is based on the earlier [Danilchenko 2011].

[Desai 2016] Desai, Aditya, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, and Sailesh R Subhajit Roy. "Program Synthesis Using Natural Language." 2016 IEEE/ACM 38th IEEE International Conference on Software Engineering.

Abstract: "Interacting with computers is a ubiquitous activity for millions of people. Repetitive or specialized tasks often require creation of small, often one-off, programs. End-users struggle with learning and using the myriad of domain-specific languages (DSLs) to effectively accomplish these tasks. We present a general framework for constructing program synthesizers that take natural language (NL) inputs and produce expressions in a target DSL. The framework takes as input a DSL definition and training data consisting of NL/DSL pairs. From these it constructs a synthesizer by learning optimal weights and classifiers (using NLP features) that rank the outputs of a keyword-programming based translation. We applied our framework to three domains: repetitive text editing, an intelligent tutoring system, and flight information queries. On 1200+ English descriptions, the respective synthesizers rank the desired program as the top-1 and top- 3 for 80% and 90% descriptions respectively."

Note: This is from Microsoft Research Redmond and IIT Kanpur.

[Fife 1987] Fife, Dennis W., Kevin Campbell, John Chludzinki, Nelson Corcoran, Carlos Gonzalez, J. Bret Michael, Edgar Sibley, David Wheeler, and Christine Youngblut, October 1987, *Evaluation of Computer-Aided System Design Tools for SDI Battle Management/C3 Architecture Development*, IDA Paper P-2062, Approved for public release, distribution unlimited.

Abstract: "This IDA Paper was prepared at the request of the Strategic Defense Initiative Organization. The paper documents the findings of an evaluation on the capabilities of certain computer software/computer-aided software engineering (CASE) tools to provide computer-aided graphic design of Battle Management/C3 for the SDIO. Each tool (of five selected on the basis of the best available at this time) was installed at IDA. After training by vendor tool staff, an IDA team, using a hands-on design exercise determined the merits of the tools for SDI application. A comparative summary of the tools is given relative to envisaged SDI requirements and an extensive questionnaire is answered for each."

[Gaunt 2016] Gaunt, Alexander L., Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. "Terpret: A Probabilistic Programming Language for Program Induction." *CoRR*, abs/1608.04428, 2016. URL http://arxiv.org/abs/1608.04428.

[Gershgorn 2017] Gershgorn, Dave. "Microsoft's AI Is Learning to Write Code by Itself, Not Steal It." 2017-03-01. https://qz.com/920468/artificial-intelligence-created-by-microsoft-and-university-of-cambridge-is-learning-to-write-code-by-itself-not-steal-it/

> This news article summarizes [Balog 2017], saying that "Microsoft and Cambridge built an algorithm capable of writing code that would solve simple math problems. The algorithm, named DeepCoder, would be able to augment its own ability by also looking at potential combinations of code for how a problem could be solved."

[Gulia 2016] Gulia, Sarita, and Tanupriya Choudhury, "An Efficient Automated Design to Generate UML Diagram From Natural Language Specifications," 2016, 2016 6th International Conference - Cloud System and Big Data Engineering (Confluence).

> Abstract: "The foremost problem that arises in the Software Development Cycle is during the Requirements specification and analysis. Errors that are encountered during the first phase of the cycle migrate to other phases too which in turn results in the most costly process than the original specified process. The reason is that the specifications of software requirements are termed in the Nature Language Format. One can easily transform the requirements specified into computer model using UML. To minimize the errors that arise in the existing system, we have proposed a new technique that enhances the generation of UML models through Natural Language requirements, which can easily provide automatic assistance to the developers. The main aim of our paper is to focus on the production of Activity Diagram and Sequence Diagram through Natural Language Specifications. Standard POS tagger and parser analyze the input i.e., requirements in English language given by the users and extract phrases, activities, etc. from the text specifies. The technique is beneficial as it reduces the gap between informal natural language and the formal modeling language. The input is the requirements laid down by the users in English language. Some stages like pre-processing, part of speech (POs), tagging, parsing, phrase identification and designing of UML diagrams occur along with the input. The application and its framework is developed in Java and it is tested on by implementing on a few technical documents."

[Gulwani 2011] Gulwani, Sumit, "Automating String Processing in Spreadsheets Using Input-Output Examples", *POPL*, 2011, https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/popl11-synthesis.pdf

> Abstract: "We describe the design of a string programming/expression language that supports restricted forms of regular expressions, conditionals and loops. The

language is expressive enough to represent a wide variety of string manipulation tasks that end-users struggle with. We describe an algorithm based on several novel concepts for synthesizing a desired program in this language from input-output examples. The synthesis algorithm is very efficient taking a fraction of a second for various benchmark examples. The synthesis algorithm is interactive and has several desirable features: it can rank multiple solutions and has fast convergence, it can detect noise in the user input, and it supports an active interaction model wherein the user is prompted to provide outputs on inputs that may have multiple computational interpretations.
The algorithm has been implemented as an interactive add-in for Microsoft Excel spreadsheet system. The prototype tool has met the golden test - it has synthesized part of itself, and has been used to solve problems beyond author's imagination."

[Gulwani 2012] Sumit Gulwani, William R. Harris, and Rishabh Singh, "Spreadsheet Data Manipulation Using Examples," *Communications of the ACM*, August 2012, Vol. 55, no. 8. https://people.csail.mit.edu/rishabh/papers/cacm12.pdf

Abstract: "Millions of computer end users need to perform tasks over large spreadsheet data, yet lack the programming knowledge to do such tasks automatically. We present a programming by example methodology that allows end users to automate such repetitive tasks. Our methodology involves designing a domain-specific language and developing a synthesis algorithm that can learn programs in that language from user-provided examples. We present instantiations of this methodology for particular domains of tasks: (a) syntactic transformations of strings using restricted forms of regular expressions, conditionals, and loops, (b) semantic transformations of strings involving lookup in relational tables, and (c) layout transformations on spreadsheet tables. We have implemented this technology as an add-in for the Microsoft Excel Spreadsheet system and have evaluated it successfully over several benchmarks picked from various Excel help forums."

Extract: "We then develop the following:

Domain-specific language: We design a domain-specific language L that is expressive enough to capture several real-world tasks in the domain, but also restricted enough to enable efficient learning from examples.

Data structure for representing consistent programs: The number of programs in L that are consistent with a given set of input–output examples can be huge. We define a data structure D based on a version-space algebra to succinctly represent a large set of such programs.

Algorithm for synthesizing consistent programs: Our synthesis algorithm for language L applies two key procedures: (i) Generate learns the set of all programs,

represented using data structure D, that are consistent with a given single example. (ii) Intersect intersects these sets (each corresponding to a different example).

Ranking: We develop a scheme that ranks programs, preferring programs that are more general. Each ranking scheme is inspired by Occam's razor, which states that a smaller and simpler explanation is usually the correct one. We define a partial order relationship between programs to compare them. Any partial order can be used that efficiently orders programs represented in the version-space algebra used by the data structure D. Such an order can be applied to efficiently select the top-ranked programs from among a set represented using D. The ranking scheme can also take into account any test inputs provided by the user (i.e., new additional inputs on which the user may execute a synthesized program). A program that is undefined on any test input or generates an output whose characteristics are different from that of training outputs can be ranked lower."

This paper builds on [Gulwani 2011].

[Gulwani 2015] Gulwani, Sumit, José Hernández-Orallo, Emanuel Kitzelmann, Stephen H. Muggleton, Ute Schmid, and Benjamin Zorn, "Inductive Programming Meets the Real World," *Communications of the ACM* (CACM), November 2015, Vol. 58 No. 11, https://dl.acm.org/citation.cfm?id=2838899.2736282

[Hellendoorn 2017] Hellendoorn, Vincent J., and Premkumar Devanbu. "Are Deep Neural Networks the Best Choice for Modeling Source Code?" ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany.

Abstract: "Current statistical language modeling techniques, including deeplearning based models, have proven to be quite effective for source code. We argue here that the special properties of source code can be exploited for further improvements. In this work, we enhance established language modeling approaches to handle the special challenges of modeling source code, such as: frequent changes, larger, changing vocabularies, deeply nested scopes, etc. We present a fast, nested language modeling toolkit specifically designed for software, with the ability to add & remove text, and mix & swap out many models. Specifically, we improve upon prior cache-modeling work and present a model with a much more expansive, multi-level notion of locality that we show to be well-suited for modeling software. We present results on varying corpora in comparison with traditional N-gram, as well as RNN, and LSTM deep-learning language models, and release all our source code for public use. Our evaluations suggest that carefully adapting N-gram models for source code can yield performance that surpasses even RNN and LSTM based deep-learning models."

[Hardesty 2015] Hardesty, Larry, "Probabilistic Programming Does in 50 Lines of Code What Used to Take Thousands," 2015-04-13, pys.org, https://phys.org/news/2015-04-probabilistic-lines-code-thousands.html

[Imam 2014] Imam, Ayad Tareq, Thamer Rousan, and Shadi Aljawarneh (all Faculty of IT; Al-Isra University Amman; Jordan), "An Expert Code Generator using Rule-Based and Frames Knowledge Representation Techniques," 2014 5th International Conference on Information and Communication Systems (ICICS)

Abstract: "This paper aims to demonstrate the development of an expert code generator using rule-based and frames knowledge representation techniques (ECG-RF). The ECG-RF system presented in this paper is a passive code generator that carries out the task of automatic code generation in fixed structure software. To develop an ECG-RF system, the artificial intelligence (AI) of rule-based system and frames knowledge representation techniques was applied to a code generation task. ECG-RF fills a predefined frame of a certain fixed-structure program with code chunks retrieved from ECG-RF's knowledge base. The filling operation is achieved by ECG-RF's inference engine and is guided by the information collected from the user via a graphic user interface (GUI). In this paper, an ECG-RF system for generating a device driver program is presented and implemented with VBasic software. The results show that the ECG-RF design concept is reasonably reliable."

Notes: The code/data collection engine "begins by loading the frame of the SW to be filled, searching for proper code fragments or data, and then assigns the selected code fragments or data to a slot in the SW frame. The search process is guided by the attributes of the slot to be filled. These attributes are collected one by one…. The collection is achieved via a wizard technique, which asks the user a question and collects the answer. This answer is used to select the next question that relates to a new property in order to accumulate properties. This accumulation of properties is an implementation of the forward chaining method... For testing purposes, device driver software has been selected as a case study in this research, and for simplicity the DOS device driver will be used." Assembly language is generated.

[Jaderberg 2019] "Human-Level Performance in 3D Multiplayer Games with Population-Based Reinforcement Learning." *Science* 31 May 2019: Vol. 364, Issue 6443, pp. 859–865 DOI: 10.1126/science.aau6249.

Abstract: "Reinforcement learning (RL) has shown great success in increasingly complex single-agent environments and two-player turn-based games. However, the real world contains multiple agents, each learning and acting independently to cooperate and compete with other agents. We used a tournament-style evaluation to demonstrate that an agent can achieve human-level performance in a three-dimensional multiplayer first-person video game, Quake III Arena in Capture the

Flag mode, using only pixels and game points scored as input. We used a two-tier optimization process in which a population of independent RL agents are trained concurrently from thousands of parallel matches on randomly generated environments. Each agent learns its own internal reward signal and rich representation of the world. These results indicate the great potential of multiagent reinforcement learning for artificial intelligence research."

[Karpathy 2015] Karpathy, Andrej. "The Unreasonable Effectiveness of Recurrent Neural Networks." May 21, 2015. http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Extract: "There's something magical about Recurrent Neural Networks (RNNs). I still remember when I trained my first recurrent network…Within a few dozen minutes of training my first baby model (with rather arbitrarily-chosen hyperparameters) started to generate very nice looking descriptions of images that were on the edge of making sense. Sometimes the ratio of how simple your model is to the quality of the results you get out of it blows past your expectations… I'm training RNNs all the time and I've witnessed their power and robustness many times, and yet their magical outputs still find ways of amusing me. This post is about sharing some of that magic with you."

[Kimber 2012] Kimber, Timothy, *Learning Definite and Normal Logic Programs by Induction on Failure*, PhD dissertation, Imperial College London, Department of Computing, 2012, https://spiral.imperial.ac.uk:8443/handle/10044/1/9961

Abstract: "This thesis presents two novel inductive logic programming (ILP) approaches, based on the notion of a connected theory. A connected theory contains clauses that depend on one another, either directly or via clauses in the background knowledge. Generalisation of such a theory is proved to be a sound and complete method for learning definite ILP hypotheses. The Induction on Failure (IOF) proof procedure, based on the connected theory generalisation method, adds secondary examples into the hypothesis, and generates auxiliary clauses to explain them. These concepts, novel to IOF, address the issues of incompleteness present in previous definite ILP methods.

The concept of the connected theory is also applied to the non-monotonic, normal program setting. Thus, the method of generalisation of a normal connected theory is presented. Fundamental to this is the assertion that a partial non-monotonic hypothesis must include both positive and negative information, which the general hypothesis should preserve. This has resulted in, as far as the author is aware, the most complete semantic characterisation available of non-monotonic ILP using a bridge formula. It is proved that generalisation of such a formula to a set of completed definitions is a sound method of generating normal program hypotheses. In the course of establishing a completeness result for this latter approach, the

semantics of the supported consequences of a normal program are defined, and the support tree method is presented and shown to be a sound and complete proof procedure for supported consequences. Using these results, it is shown that, for functionfree programs, any correct hypothesis for which the examples are supported consequences of the learned program can be derived via a normal connected theory."

[Klein 2015a] Klein, John, 2015-05-11, "Model Driven Engineering: Automatic Code Generation and Beyond," *Software Engineering Institute (SEI) Blog*, https://insights.sei.cmu.edu/sei_blog/2015/05/model-driven-engineering-automatic-code-generation-and-beyond.html

[Klein 2015b] Klein, John, Harry Levinson, and Jay Marchetti, 2015-05, "Model Driven Engineering: Automatic Code Generation and Beyond," *Software Engineering Institute (SEI), Technical Report* CMU/SEI-2015-TN-005, https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=435414

Abstract: "Increasing consideration of model-driven engineering (MDE) tools for software development efforts means that acquisition executives must more often deal with the following challenge: Vendors claim that by using MDE tools, they can generate software code automatically and achieve high developer productivity. However, MDE consists of more than code generation tools; it is also a software engineering approach that can affect the entire lifecycle of a system from requirements gathering through sustainment. This report focuses on the application of MDE tools for automatic code generation when acquiring systems built using these software development tools and processes. The report defines some terminology used by MDE tools and methods, emphasizing that MDE consists of both tools and methods that must align with overall acquisition strategy. Next, it discusses how the use of MDE for automatic code generation affects acquisition strategy and introduces new risks to the program. It then offers guidance on selecting, analyzing, and evaluating MDE tools in the context of risks to an organization's acquisition effort throughout the system lifecycle. Appendices provide a questionnaire that an organization can use to gather information about vendor tools along with criteria for evaluating tools mapped to the questionnaire that relate to acquisition concerns."

The introduction states that, "The simple answer might be, "yes, the state of the practice can achieve productivity rates of thousands of function points and millions of lines of code per person-month using MDE tools for automatic code generation." The complicated reality is that MDE consists of more than code generation tools; it is a software engineering approach that can impact the entire lifecycle from requirements gathering through sustainment. While one can make broad generalizations about these methods and tools, it is more useful to consider them in

the context of a particular system acquisition. Aligning MDE methods and tool capabilities with the system acquisition strategy can improve system quality, reduce time to field, and reduce sustainment cost. On the other hand, when MDE methods and tools do not align with the acquisition strategy, using them can result in increased risk and cost in development and sustainment.

We use [model-driven engineering (MDE)] to refer descriptively to a software development approach that treats models as the primary artifacts created and used by software lifecycle processes…. There is no single model of a system; instead, there can (and should) be multiple models. … we have to maintain consistency between models and the implementation, consistency among the models across tasks, and consistency among the models throughout the lifecycle to correctly and completely represent our system. Each model is represented using a modeling language that has a graphical representation, a textual representation, or both. Typical modeling languages include the standards-based Unified Modeling Language (UML) and Architecture Analysis and Design Language (AADL) and proprietary languages such as the Integranova Model Execution System (M.E.S.)."

Note that terms such as "artificial intelligence" and "machine learning" never occur in this paper and that the term "knowledge" always refers to organizational knowledge.

[Kneuss 2015] Kneuss, E., M. Koukoutos, and V. Kuncak. "Deductive Program Repair." In *Computer-Aided Verification: 27th International Conference, CAV 2015 Proceedings Part II*.

[Li 2018] Li, Jian, Yue Wang, Michael R. Lyu, and Irwin King, "Code Completion with Neural Attention and Pointer Networks," (The Chinese University of Hong Kong, China). attenarXiv:1711.09573v2 [cs.CL] 14 May 2018

Abstract: "Intelligent code completion has become an essential research task to accelerate modern software development. To facilitate effective code completion for dynamically-typed programming languages, we apply neural language models by learning from large codebases, and develop a tailored attention mechanism for code completion. However, standard neural language models even with attention mechanism cannot correctly predict the out-of-vocabulary (OoV) words that restrict the code completion performance. In this paper, inspired by the prevalence of locally repeated terms in program source code, and the recently proposed pointer copy mechanism, we propose a pointer mixture network for better predicting OoV words in code completion. Based on the context, the pointer mixture network learns to either generate a withinvocabulary word through an RNN component, or regenerate an OoV word from local context through a pointer component.

Experiments on two benchmarked datasets demonstrate the effectiveness of our attention mechanism and pointer mixture network on the code completion task."

Extracts:

"Intelligent code completion is one of the most useful features in IDEs, which suggests next probable code tokens... Traditionally, code completion relies heavily on compile-time type information... Thus, it works well for statically-typed languages such as Java. Yet code completion is harder and less supported for dynamically-typed languages like JavaScript and Python, due to the lack of type annotations."

"To render effective code completion for dynamically-typed languages, recently, researchers turn to learning-based language models... In particular, neural language models such as Recurrent Neural Networks (RNNs) can capture sequential distributions and deep semantics, hence become very popular. However, these standard neural language models are limited by the so-called hidden state bottleneck: all the information about current sequence is compressed into a fixed-size vector. The limitation makes it hard for RNNs to deal with long-range dependencies, which are common in program source code such as a class identifier declared many lines before it is used."

"Attention mechanism [Bahdanau et al., 2014] provides one solution to this challenge. With attention, neural language models learn to retrieve and make use of relevant previous hidden states, thereby increasing the model's memorization capability and providing more paths for back-propagation. To deal with long-range dependencies in code completion, we develop a tailored attention mechanism which can exploit the structure information on program's abstract syntax tree (AST)..."

"But even with attention, there is another critical issue called (the0 unknown word problem... a common practice is to build the vocabulary with only K most frequent words in the corpus... In code completion, simply recommending an (Unknown) token offers no help to the developers... For our code completion task, we observe that when writing programs, developers tend to repeat locally... when predicting such unknown words, our model can learn to choose one location in local context and copy the word at that location as our prediction."

"In this paper, to facilitate effective code completion, we propose a pointer mixture network, which can predict next word by either generating one from the global vocabulary or copying a word from the local context. For the former, we apply a standard RNN with attention, which we call the global RNN component. For the latter, we employ a pointer network which we call the local pointer component. Actually the two components share the same RNN architecture and attention scores. Our pointer mixture network is a weighted combination of the two components. At

each prediction, a switcher is learned based on the context information, which can guide the model to choose one component for generating the next word. In this way, our model learns when and where to copy an OoV word from the local context as the final prediction."

"The main contributions of this work are as follows:

We propose a pointer mixture network for better predicting OoV words in code completion, which learns to generate next word from either the global vocabulary or the local context.

We develop an attention mechanism for code completion, which makes use of the AST structure information (specially, the parent-children information).

We evaluate our models on two benchmarked datasets (JavaScript and Python). The experimental results show great improvements upon the state-of-the-arts (sic)"

[Laird 2017] Laird, John E., Kevin Gluck, John Anderson, Kenneth D. Forbus, Odest Chadwicke Jenkins, Christian Lebiere, Dario Salvucci, Matthias Scheutz, Andrea Thomaz, Greg Trafton, Robert E. Wray, Shiwali Mohan, and James R. Kirk, "Interactive Task Learning," July/August 2017, *IEEE Intelligent Systems*, IEEE Computer Society, DOI 1541-1672/17, http://web.eecs.umich.edu/~soar/sitemaker/docs/pubs/Laird_et_al_InteractiveTaskLearning_IEEE_IntelligentSystems_2017.pdf

Abstract: "In interactive task learning [(ITL)], an agent actively tries to learn the actual definition of a task through natural interaction with a human instructor, not just how to perform a task better."

Extract: This paper attempts "to provide an overview of the landscape of interactive task learning, including its definition, the associated desiderata for evaluating future progress, a review of previous research in related areas, and potential application areas. Interactive task learning is a constellation of approaches and research problems that if realized, holds the promise of dramatically changing the way we develop and extend the capabilities of intelligent agents. No longer will we need to rely on programmers to anticipate all of the potential tasks that our agents will perform."

Note that this is not a demonstration of some completed research, but an argument for the establishment of a subfield.

[Ling 2016] Ling, Wang, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiský, Andrew Senior, Fumin Wang, and Phil Blunsom. "Latent Predictor Networks for Code Generation." In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, 2016.* https://arxiv.org/abs/1603.06744

Abstract: “Many language generation tasks require the production of text conditioned on both structured and unstructured inputs. We present a novel neural network architecture which generates an output sequence conditioned on an arbitrary number of input functions. Crucially, our approach allows both the choice of conditioning context and the granularity of generation, for example characters or tokens, to be marginalised, thus permitting scalable and effective training. Using this framework, we address the problem of generating programming code from a mixed natural language and structured specification. We create two new data sets for this paradigm derived from the collectible trading card games Magic the Gathering and Hearthstone. On these, and a third preexisting corpus, we demonstrate that marginalising multiple predictors allows our model to outperform strong benchmarks.”

[Long 2016] Long, Fan, and Martin Rinard. “Automatic Patch Generation by Learning Correct Code.” 2016. MIT CSAIL. https://people.csail.mit.edu/fanl/papers/prophet-popl16.pdf

Abstract: “We present Prophet, a novel patch generation system that works with a set of successful human patches obtained from open-source software repositories to learn a probabilistic, application-independent model of correct code. It generates a space of candidate patches, uses the model to rank the candidate patches in order of likely correctness, and validates the ranked patches against a suite of test cases to find correct patches. Experimental results show that, on a benchmark set of 69 real-world defects drawn from eight open-source projects, Prophet significantly outperforms the previous state-of-the-art patch generation system.”

Extract: “We present Prophet, a new generate-and-validate patch generation system for repairing defects in large, real-world applications. To the best of our knowledge, Prophet is the first system to learn a probabilistic model of correct code. Prophet uses this model to automatically generate correct patches that repair defects in incorrect applications… Generate-and-validate systems start with a program and a suite of test cases, at least one of which exposes a defect in the program. They then generate a space of candidate patches and search this space to find plausible patches that produce correct outputs for all test cases in the test suite. Unfortunately, the presence of plausible but incorrect patches (which produce correct outputs for all of the test cases in the test suite but incorrect outputs for other inputs) has complicated the ability of previous generate-and-validate systems to find correct patches within the (potentially quite large) space of plausible but incorrect patches. Prophet uses its learned model of correct code to rank the patches in its search space, with the goal of obtaining a correct patch as the first (or one of the first few) patches to validate.”

[Memon 2001] Memon, Atif M., Martha E. Pollack, and Mary Lou Soffa. “Hierarchical GUI Test Case Generation Using Automated Planning.” *IEEE Transactions on Software Engineering*, Vol. 27, No. 2, February 2001.

Abstract: “The widespread use of GUIs for interacting with software is leading to the construction of more and more complex GUIs. With the growing complexity come challenges in testing the correctness of a GUI and its underlying software. We present a new technique to automatically generate test cases for GUIs that exploits planning, a well-developed and used technique in artificial intelligence. Given a set of operators, an initial state, and a goal state, a planner produces a sequence of the operators that will transform the initial state to the goal state. Our test case generation technique enables efficient application of planning by first creating a hierarchical model of a GUI based on its structure. The GUI model consists of hierarchical planning operators representing the possible events in the GUI. The test designer defines the preconditions and effects of the hierarchical operators, which are input into a plan-generation system. The test designer also creates scenarios that represent typical initial and goal states for a GUI user. The planner then generates plans representing sequences of GUI interactions that a user might employ to reach the goal state from the initial state. We implemented our test case generation system, called Planning Assisted Tester for grapHical user interface Systems (PATHS) and experimentally evaluated its practicality and effectiveness. We describe a prototype implementation of PATHS and report on the results of controlled experiments to generate test cases for Microsoft's WordPad.”

Notes: This approach is designed to create test cases, not application code. However, test cases are often created by writing code, so technically this is a way to replace writing code.

[Moderez 2016] Molderez, Tim, and Coen De Roover, “Automated Generalization and Refinement of Code Templates with EKEKO/X,” 2016, In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, https://ieeexplore.ieee.org/abstract/document/7476695

Abstract: “Code templates are an intuitive means to specify source code snippets of interest, such as all instances of a bug, groups of snippets that need to be refactored or transformed, or instances of design patterns. While intuitive, it is not always straightforward to write a template that produces only the desired matches. A template could produce either more snippets than desired, or too few. To assist the users of EKEKO/X, our template-based search and transformation tool for Java, we have extended it with two components: The first is a suite of mutation operators that simplifies the process of modifying templates. The second is a system that can automatically suggest a sequence of mutations to a given template, such that it matches only with a set of desired snippets. In this tool paper, we highlight the key

design decisions in implementing these two components of EKEKO/X, and demonstrate their use by walking through an example sequence of mutations suggested by the system."

Notes: Their search-based system can "automatically generalize and refine a template group, such that it matches only with a given set of desired code snippets. This system is based on a single objective genetic search algorithm, and it makes use of our suite of mutation operators…. the [genetic] algorithm consists of a loop that 'evolves' a set of template groups, with the aim of approaching a solution template, with a fitness value of 1. The fitness value indicates 'how good' a template group is…"

[More 2012] More, Priyanka, and Rashmi Phalnikar. "Generating UML Diagrams from Natural Language Specifications." *International Journal of Applied Information Systems*, Volume 1, No. 8, April 2012.

Abstract: "The process of generating UML Diagrams from natural language specification is a highly challenging task. This paper proposes a method and tool to facilitate the requirements analysis process and extract UML diagrams from textual requirements using natural language processing (NLP) and Domain Ontology techniques. Requirements engineers analyze requirements manually to understand the scope of the system. The time spent on the analysis and the low quality of human analysis justifies - the need of a tool for better understanding of the system. "Requirement analysis to Provide Instant Diagrams (RAPID)" is a desktop tool to assist requirements analysts and Software Engineering students to analyze textual requirements, finding core concepts and its relationships, and extraction UML diagrams. The evaluation of RAPID system is in the process and will be conducted through two forms of evaluation, experimental and expert evaluation."

Comments: This paper describes RAPID, which examines text to identify classes, their attributes, and their relationships and use the info to create a UML class diagram. The approach is not very flexible, and it is limited in what it can produce. It is possible to generate some source code from a UML class diagram.

[Murali 2018] Murali, Vijayaraghavan, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. "Neural Sketch Learning for Conditional Program Generation." ICLR 2018. 2018. https://arxiv.org/abs/1703.05698

Abstract: "We study the problem of generating source code in a strongly typed, Java-like programming language, given a label (for example a set of API calls or types) carrying a small amount of information about the code that is desired. The generated programs are expected to respect a "realistic" relationship between programs and labels, as exemplified by a corpus of labeled programs available during training. Two challenges in such conditional program generation are that the

generated programs must satisfy a rich set of syntactic and semantic constraints, and that source code contains many low-level features that impede learning. We address these problems by training a neural generator not on code but on program sketches, or models of program syntax that abstract out names and operations that do not generalize across programs. During generation, we infer a posterior distribution over sketches, then concretize samples from this distribution into type-safe programs using combinatorial techniques. We implement our ideas in a system for generating API-heavy Java code, and show that it can often predict the entire body of a method given just a few API calls or data types that appear in the method."

Note: This describes the BAYOU system, publicly available at https://github.com/capergroup/bayou under the Apache 2.0 open source software license.

[Narawita 2016] Narawita, Chamitha Ramal, and Kaneeka Vidanage (both Department of Computer Science, Informatics Institute of Technology, Sri Lanka), "UML Generator – An Automated System for Model Driven Development," *2016 International Conference on Advances in ICT for Emerging Regions (ICTer)*, pp. 250–256.

Abstract: "This research mainly focused on automation of Unified Modeling Language (UML) diagrams from the analyzed requirement text using Natural Language Processing (NLP). The proposed system is an efficient and accurate way to obtain elements of the use case and class diagrams from proposed methods. This research mainly focuses on the design phase of a software. Nowadays everybody needs a quick and reliable service. It was needed to have some sort of quick, accurate and intelligent software for generating UML based documentations to save time and budget of both the user and system analyst."

Comments: This UML Generator "generates use case and class diagram by analyzing the input text." It uses SharpNLP to do natural language processing, coupled with a simple ML approach to identify the key features. It was tested "with more than twenty (20) scenarios and it has an accuracy level of around 70%." This was an undergraduate research project.

[Nelson 2006] Nelson, Graham. 2006-04-10. "Natural Language, Semantic Analysis and Interactive Fiction." http://www.inform7.com/learn/documents/WhitePaper.pdf

Extract: "This is an account of theoretical issues which came out, almost unbidden, from a practical test of the following hypothesis: that the natural language in which to write interactive fiction (IF) is natural language. IF is a form of creative writing [where] the author creates an imaginary textual world which can actively be explored by a "reader", or "player", directing the actions of a protagonist. Such works have hitherto been created as if computer programs, using specially adapted programming languages (see for instance Nelson (2001)), but the Inform 7 project

aims to replace such syntax with natural language: specifically, a subset of English…"

[Neubig 2016] Graham Neubig. "Survey of Methods to Generate Natural Language from Source Code." 2016. http://www.languageandcode.org/nlse2015/neubig15nlse-survey.pdf.

[Nguyen 2013] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. "Semfix: Program Repair via Semantic Analysis." In *Proceedings of the 2013 International Conference on Software Engineering, ICSE '13'*, pp. 772–781, Piscataway, NJ, USA, 2013. IEEE Press.

[Njonko 2012] Njonko, Paul Brillant Feuto, and Centre Tesnière (both at Centre Tesnière, University of Franche-Comté, Besancon, France), "From Natural Language Business Requirements to Executable Models via SBVR," 2012 International Conference on Systems and Informatics (ICSAI 2012).

Abstract: "This paper presents a methodology for transforming business rules (BR) written in natural language (NL) such as English into a set of executable models as Unified Modeling Language (UML), Structured Query language (SQL), etc. As the direct automatic transformation of NL specifications to executable models is very difficult due to the inherent ambiguities of NL, this methodology aims at using the Semantics of Business Vocabulary and Business Rules (SBVR) as an intermediate model front-ended by Micro-Systemic Linguistic Analysis (MSLA) because of their mathematical underpinnings. SBVR is a Semantic Metamodel (SMM) introduced by the Object Management Group (OMG) for specifying semantic models of business using NL. SBVR is not only easy to process by machine since it is grounded in formal logic, but it is also easy to understand both by software developers and other stakeholders. Given that SBVR is fully integrated in OMG's Model Driven Architecture (MDA) and behaves as a Computational Independent Model (CIM), our approach advocates model transformation which is the key constituent of the MDA standard."

[OMG 2017] Object Management Group (OMG). "Semantics of Business Vocabulary and Business Rules (SBVR)." 2017. Version 1.4. https://www.omg.org/spec/SBVR/About-SBVR/

Extract: "This specification defines the vocabulary and rules… for documenting the semantics of business vocabularies and business rules for the exchange of business vocabularies and business rules among organizations and between software tools. This specification is interpretable in predicate logic with a small extension using modal operators. It supports linguistic analysis of text for business vocabularies and business rules, with the linguistic analysis itself being outside the scope of this specification... The SBVR specification is applicable to the domain of business

vocabularies and business rules of all kinds of business activities in all kinds of organizations. It provides an unambiguous, meaning-centric, multilingual, and semantically rich capability for defining meanings of the language used by people in an industry, profession, discipline, field of study, or organization. This specification is conceptualized optimally for business people rather than automated processing. It is designed to be used for business purposes, independent of information systems designs to serve these business purposes…"

SBVR is primarily a modelling system that can be used for many things, including implementation in information systems, and its rules are represented using XML. However, its Annex A defines "SBVR Structured English" as a rigid form of structured English and a mapping to the underlying model. As Annex A states: "The most common means of expressing definitions and business rules is through statements, not diagrams. While diagrams are helpful for seeing how concepts are related, they are impractical as a primary means of defining vocabularies and expressing business rules. This specification defines an English vocabulary for describing vocabularies and stating rules. There are many different ways that this vocabulary and other English vocabularies described using SBVR can be combined with common English words and structures to express definitions and statements. However expressed, the semantics of definitions and rules can be formally represented in terms of the SBVR vocabulary and, particularly, in terms of logical formulations (the SBVR conceptualization of formal logic). This annex describes one such way of using English that maps mechanically to SBVR concepts. It is not meant to offer all of the variety of common English, but rather, it uses a small number of English structures and common words to provide a simple and straightforward mapping."

Examples of SBVR Structured English (from its specification) are "It is obligatory that each rental car is owned by exactly one branch," "A rental must have at most three additional drivers," "A car must be assigned to a rental before the pick-up date of the rental," and "A rental must be guaranteed by a credit card before a car is assigned to the rental."

[Oord] van den Oord, Aäron, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. *Wavenet: A Generative Model for Raw Audio*.

Abstract: "This paper introduces WaveNet, a deep neural network for generating raw audio waveforms. The model is fully probabilistic and autoregressive, with the predictive distribution for each audio sample conditioned on all previous ones; nonetheless we show that it can be efficiently trained on data with tens of thousands of samples per second of audio. When applied to text-to-speech, it yields state-of-the-art performance, with human listeners rating it as significantly more natural

sounding than the best parametric and concatenative systems for both English and Chinese. A single WaveNet can capture the characteristics of many different speakers with equal fidelity, and can switch between them by conditioning on the speaker identity. When trained to model music, we find that it generates novel and often highly realistic musical fragments. We also show that it can be employed as a discriminative model, returning promising results for phoneme recognition."

This can generate audio from text; some of its ideas might also be applicable to generating code from text.

[Parnas 1985] Parnas, David. "Software Aspects of Strategic Defense Systems." *Communications of the ACM*. Volume 28, Number 12, December 1985.

[Pradel 2017] Pradel, Michael, and Koushik Sen. "Deep Learning to Find Bugs." Technical Report TUD-CS-2017-0295. TU Darmstadt, Department of Computer Science. November 2017. http://mp.binaervarianz.de/DeepBugs_TR_Nov2017.pdf

[Pradel 2018] Pradel, Michael, and Koushik Sen. "DeepBugs: A Learning Approach to Name-Based Bug Detection." *Proc. ACM Program. Lang. 2, OOPSLA*, Article 147 (November 2018). https://doi.org/10.1145/3276517 https://software-lab.org/publications/oopsla2018_DeepBugs.pdf

Abstract: "Natural language elements in source code, e.g., the names of variables and functions, convey useful information. However, most existing bug detection tools ignore this information and therefore miss some classes of bugs. The few existing name-based bug detection approaches reason about names on a syntactic level and rely on manually designed and tuned algorithms to detect bugs. This paper presents DeepBugs, a learning approach to name-based bug detection, which reasons about names based on a semantic representation and which automatically learns bug detectors instead of manually writing them. We formulate bug detection as a binary classification problem and train a classifier that distinguishes correct from incorrect code. To address the challenge that effectively learning a bug detector requires examples of both correct and incorrect code, we create likely incorrect code examples from an existing corpus of code through simple code transformations. A novel insight learned from our work is that learning from artificially seeded bugs yields bug detectors that are effective at finding bugs in real-world code. We implement our idea into a framework for learning-based and name-based bug detection. Three bug detectors built on top of the framework detect accidentally swapped function arguments, incorrect binary operators, and incorrect operands in binary operations. Applying the approach to a corpus of 150,000 JavaScript files yields bug detectors that have a high accuracy (between 89% and 95%), are very efficient (less than 20 milliseconds per analyzed file), and reveal 102 programming mistakes (with 68% true positive rate) in real-world code."

[Priya 2017] Priya, Renita, Xinyuan Wang, Yu Sun, and Yujie Hu. “A Deep Dive into Automatic Code Generation Using Character Based Recurrent Neural Networks.” 2017 International Conference on Computational Science and Computational Intelligence.

Abstract: “Deep Learning is an emerging field in Artificial Intelligence that uses biologically inspired neural networks to recognize patterns in the natural world. These neural networks have an amazing ability to process large amounts of data and learn from them. Recurrent Neural Networks (RNN) are used in applications involving natural language processing like text translations and text generation. This research evaluates the effectiveness of a RNN to be able to automatically generate programming code. Programming languages are different from natural languages in that they have unique structure and syntax. The goal for this research is to conduct experiments on a character RNN model with for three programming languages; Java, Python and C#, and evaluate the results by testing and analyzing the ability for the RNN to automatically produce code that is able to compile.”

Comments: This is not useful, as it merely attempts to generate code that compiles or runs rather than code that performs a specific task. Interestingly, one of the authors is from an Irvine, CA, high school.

[Rajeev 2013] Rajeev, Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa,

“Syntax-guided synthesis,” *2013 Formal Methods in Computer-Aided Design*, DOI: 10.1109/FMCAD.2013.6679385, https://ieeexplore.ieee.org/document/6679385/authors

Abstract: “The classical formulation of the program-synthesis problem is to find a program that meets a correctness specification given as a logical formula. Recent work on program synthesis and program optimization illustrates many potential benefits of allowing the user to supplement the logical specification with a syntactic template that constrains the space of allowed implementations. Our goal is to identify the core computational problem common to these proposals in a logical framework. The input to the syntax-guided synthesis problem (SyGuS) consists of a background theory, a semantic correctness specification for the desired program given by a logical formula, and a syntactic set of candidate implementations given by a grammar. The computational problem then is to find an implementation from the set of candidate expressions so that it satisfies the specification in the given theory. We describe three different instantiations of the counter-example-guided-inductive-synthesis (CEGIS) strategy for solving the synthesis problem, report on prototype implementations, and present experimental results on an initial set of benchmarks.”

Extract: “we have developed an interchange format, called SYNTH-LIB, based on the syntax of SMT-LIB2-the input format accepted by the SMT solvers (see smt-

lib.org). The input for the SyGuS problem to synthesize the function f with the specification φ1 in the theory LIA, with the grammar for the languages L1 is encoded in SYNTH-LIB as…"

[Raza 2015] Raza, Mohammad, Sumit Gulwani, and Natasa Milic-Frayling. "Compositional Program Synthesis from Natural Language and Examples." *IJCAI'15 Proceedings of the 24th International Conference on Artificial Intelligence*. pp. 792–800. Buenos Aires, Argentina — July 25–31, 2015. https://dl.acm.org/citation.cfm?id=2832249.2832359

Abstract: "Compositionality is a fundamental notion in computation whereby complex abstractions can be constructed from simpler ones, yet this property has so far escaped the paradigm of end-user programming from examples or natural language. Existing approaches restrict end users to only give holistic specifications of tasks, which limits the expressivity and scalability of these approaches to relatively simple programs in very restricted domains. In this paper we propose Compositional Program Synthesis (CPS): an approach in which tasks can be specified in a compositional manner through a combination of natural language and examples. We present a domain-agnostic program synthesis algorithm and demonstrate its application to an expressive string manipulation language. We evaluate our approach on complex tasks from online help forums that are beyond the scope of current state-of-the-art methods."

This is a result from Microsoft Research. They focus on a very narrow domain: manipulation of strings of characters.

[Reynolds 2017] Reynolds, Matt. "AI Learns to Write Its Own Code by Stealing From Other Programs." *New Scientist*. 2017-02-22. https://www.newscientist.com/article/mg23331144-500-ai-learns-to-write-its-own-code-by-stealing-from-other-programs/

This news article discusses DeepCoder — see {Balog 2017].

[Rich 1992] Rich, Charles, and Richard C. Waters, *Approaches to Automatic Programming*, Mitsubishi Electric Research Laboratories, TR92-04, July 1992. http://www.merl.com/publications/docs/TR92-04.pdf

Abstract: "This paper is an overview of current approaches to automatic programming organized around three fundamental questions that must be addressed in the design of any automatic programming system: What does the user see? How does the system work? What does the system know? As an example of a research effort in this area, we focus the Programmer´s Apprentice project."

[Riedel 2016] Riedel, Sebastian, Matko Bosnjak, and Tim Rocktäschel. “Programming with a Differentiable Forth Interpreter.” *CoRR*, abs/1605.06640, 2016. URL http://arxiv.org/abs/1605.06640.

[Rosales-Morales 2015] Rosales-Morales, Viviana Yarel, Giner Alor-Hernández, Jorge Luis García-Alcaráz, Ramón Zatarain-Cabada, and María Lucía Barrón-Estrada, “An Analysis of Tools for Automatic Software Development and Automatic Code Generation,” *Revista Facultad de Ingeniería*, Universidad de Antioquia, No. 77, pp. 75–87, 2015, http://aprendeenlinea.udea.edu.co/revistas/index.php/ingenieria/article/view/21957/20961

Abstract: “Software development is an important area in software engineering, which is why a wide range of techniques, methods, and approaches has emerged to facilitate software development automation. This paper presents an analysis and evaluation of tools for automated software development and automatic code generation in order to determine whether they meet a set of quality metrics. Diverse quality metrics were considered such as effectiveness, productivity, safety, and satisfaction in order to carry out a qualitative and quantitative evaluation. The tools evaluated are CASE tools, frameworks, and Integrated Development Environments (IDEs). The evaluation was conducted to measure not only the tools’ ability to be employed, but also their support for automated software development and automatic source code generation. The aim of this work is to provide a methodology and a brief review of the most important works to identify the main features of these works and present a comparative evaluation in qualitative and quantitative terms of quality metrics. This would provide software developers with the information they need to decide the tools that can be useful for them.”

Note that this survey does not generally identify the approaches as applying AI. The only exception is its reference to an “Artificial Coder using Artificial Intelligence” (ACAI), which we refer to as [Danilchenko2012].

[Roychoudhury 2017] Roychoudhury, Suman , Sagar Sunkle, Deepali Kholkar, and Vinay Kulkarni (Tata Consultancy Services Research, India), “From Natural Language to SBVR Model Authoring Using Structured English for Compliance Checking,” 2325-6362/17, *IEEE Computer Society*, 2017, DOI 10.1109/EDOC.2017.19, 2017 IEEE 21st International Enterprise Distributed Object Computing Conference.

Abstract: “In spite of the proliferation of the business process and data compliance checking approaches, in practice, regulatory compliance management still demands considerable manual intervention. Previous research in the field of compliance has established that the manual specification/tagging of the regulations not only fails to ensure their proper coverage but also negatively affects the turnaround time both in proving and maintaining the compliance. Our contribution is an (semi-) automated

transformation of the legal NL (English) text to SBVR Model via authoring of Structured English (SE) rules. The key benefit of our approach is the direct involvement of the domain experts to specify regulations using SE, which is close to English, rather than a formal specification language. We substantiate the approach using an example from industry regulations in banking and financial services domain."

[Saad 2019] Saad, Feras, Mcro F. Cusumano-Towner, Ulrich Schaechtle, Martin C. Rinard, and Vikash K. Mansinghka (all at the Massachusetts Institute of Technology), "Bayesian Synthesis of Probabilistic Programs for Automatic Data Modeling," *Proc. ACM Program. Lang. 3*, POPL, Article 37 (January 2019). https://doi.org/10.1145/3290350, https://dl.acm.org/citation.cfm?doid=3302515.3290350

Abstract: "We present new techniques for automatically constructing probabilistic programs for data analysis, interpretation, and prediction. These techniques work with probabilistic domain-specific data modeling languages that capture key properties of a broad class of data generating processes, using Bayesian inference to synthesize probabilistic programs in these modeling languages given observed data. We provide a precise formulation of Bayesian synthesis for automatic data modeling that identifies sufficient conditions for the resulting synthesis procedure to be sound. We also derive a general class of synthesis algorithms for domain-specific languages specified by probabilistic context-free grammars and establish the soundness of our approach for these languages. We apply the techniques to automatically synthesize probabilistic programs for time series data and multivariate tabular data. We show how to analyze the structure of the synthesized programs to compute, for key qualitative properties of interest, the probability that the underlying data generating process exhibits each of these properties. Second, we translate probabilistic programs in the domain-specific language into probabilistic programs in Venture, a general-purpose probabilistic programming system. The translated Venture programs are then executed to obtain predictions of new time series data and new multivariate data records. Experimental results show that our techniques can accurately infer qualitative structure in multiple real-world data sets and outperform standard data analysis methods in forecasting and predicting new data."

[Samanta 2014] Samanta, R., O. Olivo, and E. A. Emerson. "Cost-Aware Automatic Program Repair." In *Static Analysis - 21st International Symposium, SAS 2014*, Munich, Germany, September 11–13, 2014. pp. 268–284.

[Samimi 2012] Samimi, H., M. Schäfer, S. Artzi, T. D. Millstein, F. Tip, and L. J. Hendren. "Automated Repair of HTML Generation Errors in PHP Applications Using String Constraint Solving." In *ICSE 2012*, June 2-9, 2012, Zurich, Switzerland, pp. 277–287.

[Schuman 2015] Schuman, *C. D. Neuroscience-Inspired Dynamic Architectures*, Ph.D. Dissertation, The University of Tennessee, Knoxville, TN. May, 2015. Available from http://www.lib.utk.eu/

[Sharma 2015] Sharma, Richa, Pratyoush K. Srivastava, and Kanad K. Biswas. "From Natural Language Requirements to UML Class Diagrams." 2015 IEEE Second International Workshop on Artificial Intelligence for Requirements Engineering (AIRE). pp. 1–8. IEEE.

> Abstract: "Unified Modeling Language (UML) is the most popular modeling language for analysis, design and development of the software system. There has been a lot of research interest in generating these UML models, especially class diagrams, automatically from Natural Language requirements. The interest in class diagrams can be attributed to the fact that classes represent the abstractions present in the system to be developed. However, automated generation of UML class diagrams is a challenging task as it involves lot of pre-processing or manual intervention at times. In this paper, we present dependency analysis based approach to derive UML class diagrams automatically from Natural Language requirements. We transform the requirements statements to an intermediary frame-based structured representation using dependency analysis of requirements statements and the Grammatical Knowledge Patterns. The knowledge stored in the frame-based structured representation is used to derive class diagrams using rule-based algorithm. Our approach has generated similar class diagrams as reported in earlier works based on linguistic analysis with either annotation or manual intervention. We present the effectiveness of our approach in terms of recall and precision for the case-studies presented in earlier works."

[Silver 2018] Silver, David, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. "A General Reinforcement Learning Algorithm that Masters Chess, Shogi, and Go through Self-play." *Science*, 362, pp. 1140–1144 (2018). 2018-12-07.

> Abstract: "The game of chess is the longest-studied domain in the history of artificial intelligence. The strongest programs are based on a combination of sophisticated search techniques, domain-specific adaptations, and handcrafted evaluation functions that have been refined by human experts over several decades. By contrast, the AlphaGo Zero program recently achieved superhuman performance in the game of Go by reinforcement learning from self-play. In this paper, we generalize this approach into a single AlphaZero algorithm that can achieve superhuman performance in many challenging games. Starting from random play and given no domain knowledge except the game rules, AlphaZero convincingly

defeated a world champion program in the games of chess and shogi (Japanese chess), as well as Go."

While this paper is not about generating source code per se, it is listed because it represents a stunning demonstration of modern AI capabilities. This is a single algorithm that, given only the game rules, can achieve superhuman performance in Chess, Shogi, and Go by self-play.

[Simonite 2019] Simonite, Tom, "The World's Fastest Supercomputer Breaks an AI Speed Record", *Wired*, 2019-01-31, https://www.wired.com/story/worlds-fastest-supercomputer-breaks-ai-record/

[So 2018] So, Sunbeom, and Hakjoo Oh (Korea University), "Synthesizing Pattern Programs from Examples," *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18)*, 2018

Abstract: "We describe a programming-by-example system that automatically generates pattern programs from examples. Writing pattern programs, which produce various patterns of characters, is one of the most popular programming exercises for entry level students. However, students often find it difficult to write correct solutions by themselves. In this paper, we present a method for synthesizing pattern programs from examples, allowing students to improve their programming skills efficiently. To that end, we first design a domain-specific language that supports a large class of pattern programs that students struggle with. Next, we develop a synthesis algorithm that efficiently finds a desired program by combining enumerative search, constraint solving, and program analysis. We implemented the algorithm in a tool and evaluated it on 40 exercises gathered from online forums. The experimental results and user study show that our tool can synthesize instructive solutions from 1–3 example patterns in 1.2 seconds on average."

[Tong 2016] Tong, Anh and Jaesik Choi, "Automatic Generation of Probabilistic Programming from Time Series Data", 2016, https://arxiv.org/abs/1607.00710

Abstract: "Probabilistic programming languages represent complex data with intermingled models in a few lines of code. Efficient inference algorithms in probabilistic programming languages make possible to build unified frameworks to compute interesting probabilities of various large, real-world problems. When the structure of model is given, constructing a probabilistic program is rather straightforward. Thus, main focus have been to Alearn the best model parameters and compute marginal probabilities. In this paper, we provide a new perspective to build expressive probabilistic program from continue time series data when the structure of model is not given. The intuition behind of our method is to find a descriptive covariance structure of time series data in nonparametric Gaussian

process regression. We report that such descriptive covariance structure efficiently derives a probabilistic programming description accurately."

[Volkstorf 2015] Volkstorf, Charles, "Program Synthesis from Axiomatic Proof of Correctness," 2017-01-07, https://arxiv.org/abs/1501.01363

Abstract: "Program Synthesis is the mapping of a specification of what a computer program is supposed to do, into a computer program that does what the specification says to do. This is equivalent to constructing any computer program and a sound proof that it meets the given specification.

We axiomatically prove statements of the form: program PROG meets specification SPEC. We derive 7 axioms from the definition of the PHP programming language in which the programs are to be written. For each primitive function or process described, we write a program that uses only that feature (function or process), and we have an axiom that this program meets the specification described. Generic ways to alter or combine programs, that meet known specifications, into new programs that meet known specifications, are our 7 rules of inference.

To efficiently prove statements that some program meets a given specification, we work backwards from the specification. We apply the inverses of the rules to the specifications that we must meet, until we reach axioms that are combined by these rules to prove that a particular program meets the given specification. Due to their distinct nature, typically few inverse rules apply. To avoid complex wff and program manipulation algorithms, we advocate the use of simple table maintenance and look-up functions to simulate these complexities as a prototype."

[Whalen 2016] Whalen, Daniel, "Holophrasm: A Neural Automated Theorem Prover for Higher-Order Logic," 10 Aug 2016, https://arxiv.org/abs/1608.02644

[Wheeler 2016] Wheeler, David A., and Amy E. Henninger, *State-of-the-Art Resources (SOAR) for Software Vulnerability Detection, Test, and Evaluation 2016 (aka "Software SOAR")*, Institute for Defense Analyses Paper P-8005, November 2016, https://www.acq.osd.mil/se/initiatives/init_jfac.html or https://www.ida.org/idamedia/Corporate/Files/Publications/IDA_Documents/ITSD/2017/P-8005.ashx

[Wheeler 2019] Wheeler, David A., "Secure Software Design & Programming Lecture 10: Formal Methods," 2019-05-01, https://dwheeler.com/secure-class/presentations/Secure-Software-10-Formal-Methods.ppt

[Wilkins 2018] Wilkins, Benjamin. "Sketching Interfaces Generating Code From Low Fidelity Wireframes." *AirBnb*. 2018. https://airbnb.design/sketching-interfaces/

Extract: "Sketching seemed like the natural place to start. As interface designers, sketching is an intuitive method of expressing a concept. We wanted to see how it

might look to skip a few steps in the product development lifecycle and instantly translate our sketches into a finished product. Airbnb's design system is well documented, and each component within the system has been named. We developed a working theory that … we should be able to classify the 150 components within our system and teach a machine to recognize them. We built an initial prototype [called sketch2Code] using about a dozen hand-drawn components as training data, open source machine learning algorithms, and a small amount of intermediary code to render components from our design system into the browser. We were pleasantly surprised with the result… This system has already demonstrated massive potential."

This article discusses Airbnb's sketch2Code system, which is "able to scan the mockups made by the designers and translate it into code." (summary by [Cheng 2018]). The goal was to rapidly test designs.

[Xu 2018] Xu, James Y., and Yingxu Wang (University of Calgary, Canada), "Towards and Methodology for RTPA-MATLAB Code Generation Based on Machine Learning Rules," *Proc. 2018 IEEE 17th International Conference on Cognitive Informatics and Cognitive Computing*, https://ieeexplore.ieee.org/document/8482093/

Abstract: "Autonomous program code generation by machine learning is not only an ultimate goal but also a theoretical challenge to software science and engineering. A methodology and case study for code generation based on Real-Time Process Algebra (RTPA) by machine learning are presented in this paper. It describes a machine learning approach for code generation in MATLAB based on acquired RTPA rules and formal specifications. The design and implementation of the RTPA-MATLAB code generator is introduced, which is implemented by an RTPA parser and an MATLAB code builder. The experimental case studies have demonstrated the novelty of the theories and methodologies for code generation based on machine-learnt programming rules."

Extracts: "The kernel of the RTPA-MATLAB code generator learns rules from RTPA specifications for both structure and process models in order to automatically generate code in MATLAB… the coding rules elicited from RTPA are represented in the learning engine covering rules of types, primitive operators and relational operators of programs. A finite set of basic rules for code generation is built-in as prior knowledge. The system carries out rule learning under supervision for developing its own programming knowledge base towards automatic code generation."

[Yin 2017] Yin, Pencheng, and Graham Neubig (both at Carnegie Mellon University). "A Syntactic Neural Model for General-Purpose Code Generation." *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, pp. 440–450.

Vancouver, Canada, July 30 - August 4, 2017. © 2017 Association for Computational Linguistics. https://doi.org/10.18653/v1/P17-1041

Abstract: “We consider the problem of parsing natural language descriptions into source code written in a general-purpose programming language like Python. Existing datadriven methods treat this problem as a language generation task without considering the underlying syntax of the target programming language. Informed by previous work in semantic parsing, in this paper we propose a novel neural architecture powered by a grammar model to explicitly capture the target syntax as prior knowledge. Experiments find this an effective way to scale up to generation of complex programs from natural language descriptions, achieving state-of-the-art results that well outperform previous code generation and semantic parsing approaches.”

[Zhang 2017] Zhang, Han, Tao Xu, Hongsheng Li, Shaoting Zhang, Xiaogang Wang, Xiaolei Huang, and Dimitris Metaxas. “StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks.” https://arxiv.org/abs/1612.03242

Abstract: “Synthesizing high-quality images from text descriptions is a challenging problem in computer vision and has many practical applications. Samples generated by existing text-to-image approaches can roughly reflect the meaning of the given descriptions, but they fail to contain necessary details and vivid object parts. In this paper, we propose Stacked Generative Adversarial Networks (StackGAN) to generate 256x256 photo-realistic images conditioned on text descriptions. We decompose the hard problem into more manageable sub-problems through a sketch-refinement process. The Stage-I GAN sketches the primitive shape and colors of the object based on the given text description, yielding Stage-I low-resolution images. The Stage-II GAN takes Stage-I results and text descriptions as inputs, and generates high-resolution images with photo-realistic details. It is able to rectify defects in Stage-I results and add compelling details with the refinement process. To improve the diversity of the synthesized images and stabilize the training of the conditional-GAN, we introduce a novel Conditioning Augmentation technique that encourages smoothness in the latent conditioning manifold. Extensive experiments and comparisons with state-of-the-arts on benchmark datasets demonstrate that the proposed method achieves significant improvements on generating photo-realistic images conditioned on text descriptions.”

This generates images from text; some of its ideas might also be applicable to generating code from text.

Acronyms and Abbreviations

AI	Artificial Intelligence
AST	Abstract Syntax Tree
CPS	Compositional Program Synthesis
GAN	Generative Adversarial Network
GUI	Graphical User Interface
IDE	Interactive Development Environment
IFP	Inductive Functional Programming
ILP	Inductive Logic Programming
IP	Inductive Programming
ITL	Interactive Task Learning
MDA	Model-Driven Architecture
MDE	Model-Driven Engineering
ML	Machine Learning
NLP	Natural Language Processing
NN	Neural Network
OMG	Object Management Group
OoV	Out-of-Vocabulary
OSS	Open Source Software
RAPID	Requirement Analysis to Provide Instant Diagrams
RL	Reinforcement Learning
RNN	Recurrent Neural Network
RTPA	Real-Time Process Algebra
SBVR	Semantics of Business Vocabulary and Business Rules
SyGuS	Syntax-Guided Synthesis
SyGus-Comp	Syntax-Guided Synthesis Competition
SyGuS-IF	Syntax-Guided Synthesis Input Format
StackGAN	Stacked Generative Adversarial Network
UI	User Interface
UML	Unified Modeling Language

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE (DD-MM-YY)	2. REPORT TYPE	3. DATES COVERED (From – To)
00-09-19	Final	

4. TITLE AND SUBTITLE

A Partial Survey on AI Technologies Applicable to Automated Source Code Generation

5a. CONTRACT NUMBER

HQ0034-14-D-0001

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBERS

6. AUTHOR(S)

David A. Wheeler, John D. Birdwell, Francisco L. Loaiza

5d. PROJECT NUMBER

DI-5-4630

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESSES

Institute for Defense Analyses
4850 Mark Center Drive
Alexandria, VA 22311-1882

8. PERFORMING ORGANIZATION REPORT NUMBER

NS D-10790

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Garnard W. Burnside II
Product Lead Army Enterprise Staff Management System
10119 Beach Road, BLDG 322, Room 2212, Fort Belvoir, VA 22060-5801

10. SPONSOR'S / MONITOR'S ACRONYM

PL AESMS

11. SPONSOR'S / MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

Project Leader: Francisco L. Loaiza

14. ABSTRACT

This document provides a partial survey of relatively recent research efforts reported in the literature in the area of automated source code generation (ASCG) using methods from artificial intelligence (AI) in general and machine learning (ML) in particular. The work was motivated by the fact that the Army has "identified the need to reduce costs and delivery time across the enterprise related to software generation, access, management, and sustainment." The assessments documented here support the Army Software Marketplace (ASM) Acquisition Strategy project (DI-5-4630), which calls for the "evaluation of technical options and alternatives … for standing up an enterprise-level Army Application Development Environment (ADE) that supports development for the full range of software platforms…." This paper also is intended as an input to Deliverable 4d, which calls for "a draft report on maturity and applicability of options that can support the creation of an Army ADE." In addition, the survey of emerging technologies with a sufficient degree of maturity and applicability to the workflows that will exist in the planned Army Software Factory (ASF) will inform the governance to be articulated in this phase of the ASM study.

15. SUBJECT TERMS

Artificial Intelligence (AI); Abstract Syntax Tree (AST); Compositional Program Synthesis (CPS); Generative Adversarial Network (GAN); Graphical User Interface (GUI); Interactive Development Environment (IDE); Inductive Functional Programming (IFP); Inductive Logic Programming (ILP); Inductive Programming (IP); Interactive Task Learning (ITL); Model-Driven Architecture (MDA); Model-Driven Engineering (MDE); Machine Learning (ML); Natural Language Processing (NLP); Neural Network (NN); Object Management Group (OMG); Out-of-Vocabulary (OoV); Open Source Software (OSS); Requirement Analysis to Provide Instant Diagrams (RAPID); Reinforcement Learning (RL); Recurrent Neural Network (RNN); Real-Time Process Algebra (RTPA); Semantics of Business Vocabulary and Business Rules (SBVR); Syntax-Guided Synthesis (SyGuS); Syntax-Guided Synthesis Competition ((SyGus-Comp); Syntax-Guided Synthesis Input Format (SyGuS-IF); Stacked Generative Adversarial Network (StackGAN); User Interface (UI); Unified Modeling Language (UML)

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Garnard W. Burnside II
Unclassified	Unclassified	Unclassified	Unlimited	66	19b. TELEPHONE NUMBER (Include Area Code) 703-704-4213