Creating new Program Proofs by Combining Abductive and Deductive Reasoning

Kuruvilla George Aiyankovil, Diarmuid P. O'Donoghue, Rosemary Monahan
Department of Computer Science
Maynooth University
Maynooth, Co. Kildare, Ireland
{diarmuid.odonoghue; rosemary.monahan}@mu.ie

Abstract

We describe recent work on the *Aris* system that creates and verifies new formal specifications for pre-existing source code. We describe Aris in terms of the abductive reasoning system that suggest possible specifications and then uses an existing deductive verifier to evaluate these creations. This paper focuses on the abduction system that creates new formal specifications by leveraging a small set of inspiring artefacts to augment a subset of candidate problems. This employs knowledge graphs to represent the raw data (i.e., source code), discovering latent similarities between graphs using a graph-matching process. Results are presented for the C# programming language with novel creations and its sister language called *Code Contracts*. We outline *ampliative* creativity, whereby newly created artefacts drive subsequent creative episodes beyond the initially perceived limitations. We also outline some recent work towards transferring specifications between the C# and Java programming languages.

Introduction

Formal specifications are central to adhering to safety standards and proving the correctness of mission-critical software, such as controlling nuclear reactors, vehicle and aircraft control, telecommunications infrastructure *etc*. Commercial sensitivity means that open-source specifications are not openly available, so specifications are typically created afresh. This paper addresses the challenge of creating formal specifications for existing source code, based on the *likely* intended functionality of that code.

The problem we address is to create formal specifications and proof requirements for given source code, akin to challenges in the VerifyThis (Dross, Furia, Huisman, Monahan, & Müller, 2021) series of program verification competitions. In these challenges, participants must create formal specifications, implement a solution, and formally prove that the implementation adheres to the specification. The formal specifications are written as a contract specifying the preconditions (`requires` clauses) that must hold, in order for the implementation to establish the postconditions (`ensures` clauses). The proof that the implementation adheres to the created specification is typically handled by an automated theorem prover which requires the users to provide axioms to assist the proof. These are written as `assertions, invariants, variants` alongside the source code. These specifications and axioms are often difficult for the user to create but writing them becomes easier with experience of the software domain. We are particularly interested in the creativity required in the use of both specifications and proof supports, ensuring the approaches adopted are transferable from one challenge to another (e.g., different sorting algorithms all require proof that the data is sorted and that the result is a permutation of the input data) and from one software verification tool to another (e.g., verification in Code Contracts for C# source code and Open JML for Java source code). A second challenge that software verification meets is poor uptake by industry (Huisman, Gurov, & Malkis, 2020). These challenges motivate us to investigate computational creativity to promote formal methods in the software-development process, automating some of the tasks involved in specifying existing source code. Our ARIS (Pitu, et al., 2013), (O'Donoghue, et al., 2014) system aims for professional (pro-c) creativity (Kaufman & Beghetto, 2009), comparable to professional formal-software developers.

We point out that formal specifications describe *what* an implementation achieves, while the implementation details *how* it is achieved. Specifications are often concise, describing the expected results of an operation. Formal theorem provers then verify the "what" against the "how". The *novel* outputs of Aris are the problem implementation code and new formal specifications. Implicit in any new and *useful* artefact will be the newly verified theorem, uniting information derived from the implementation and specification.

We describe previously specified code as our *inspiring set* because our objective is that Aris can use all available proof strategies to specify a given implementation. Similar implementations can require different proof strategies, depending upon the specification language and the verification tool used. Aris aspired to support the full range of available proof strategies, tailoring the chosen strategy to the source code, specification and verification tool being used.

Proceedings of the 12th International Conference on Computational Creativity (ICCC '21)
ISBN: 978-989-54160-3-5

The remainder of this paper outlines of related work before describing the structure of the Aris system, describing how its abductive and deductive reasoning systems combine to create newly verified software. We describe the very small number of inspiring artifacts and a large collection of potentially solvable problems. We then describe our results, before showing how created artefacts can serve to drive subsequent creative episodes beyond the limits of the initial creative episode. Finally, we outline possible future work.

Background

Machine learning approaches to this problem are severely impaired by the chronic lack of specifications. We analysed several thousand projects from open-source repositories (SourceForge, GitHub etc.) containing over 2,000,000 methods and did not find any Code Contracts.

Recent work on systems such as GPT-3 and Text2App have shown some ability in creating executable code from a textual description. However, our problem does not have any such textual description leading us to rely solely on source code. A major problem with previous models of abduction (O'Donoghue & Keane, 2012) concerned the unreliable nature of their inferences.

There have been several recent advances related to code completion assistants. GitHub's Copilot gives suggestions for lines of code or entire functions, taking as context any available docstrings, comments, function names, and the code itself. It's recommended that Copilot's outputs should be tested, reviewed, and vetted while in contrast, the output of Aris is evaluated by an automated theorem prover - as shall be discussed later in this paper. Interestingly, Copilot can also suggest test cases for a given implementation, however its ability to create formal specifications has not been reported. Copilot is based on Codex which is from the GPT family of language models, which is fine-tuned for code (Chen, et al., 2021). If sufficient specification were available train such a model (such as Aris can produce), these language models may become capable of producing specifications for given implementations. Despite these recent advances in code writing assistants, we are not aware of any system that can automatically add formal specifications to an existing C# implementation.

We argue that our challenge is more similar to the HR2 systems (Colton, 2012) that creates new mathematical theorems and is potentially capable of generating Prolog code. We think of problem code as containing facts and axioms that are known, with the objective of creating a new theorem (specification) likely to be useful in ensuring the correctness of that implementation. Source code and code contracts are translated into a form of first order logic and the Z3 SMT theorem prover takes the source code facts and the code contract "theorems" and attempts to verify one against the other. So, we can consider the code contracts created by our system as being akin to the theorems discovered by HR2. Aris also embodies significant differences to HR2 that created truly novel theorems, whereas Aris aims to create specification and proof supports that would be written by a competent professional formal software developer. Thus, historical H-creativity (Boden, 1992) is not required.

Figure 1 depicts the source code of a C# method for which we want to create a formal specification, highlighting a formal specification written as a precondition (i.e., requires) *Code Contract*[1]. The challenge for Aris is to create comparable formal specifications for similar implementations. The next section outlines the similarities that Aris can detect, as well as the mechanisms used to detect this similarity.

```
public char[] ReadNext(int count){
  Contract.Requires(0 <= count);
  char[] array = new char[count];
  for (int i = 0; i < count; i++)    {
    array[i] =
        this.charBuffer[this.position++]; }
  return array; }
```

Figure 1: Problem code along and a *Code Contract* (highlighted) that we wish to create.

The ARIS System

Abductive reasoning excels at proposing hypotheses based on perceived similarity to some past scenario and has become associated with creative thinking. The downside of abduction lies in the reliability of its inferences and is sometimes accused of being grossly profligate in generating inferences even where no real similarity exists. In contrast, deductive reasoning excels at deriving definite conclusions from definite premises but is sometimes associated with narrow and constrained thinking. This section outlines how Aris creates new specifications using a combination of abductive and deductive reasoning to produce novel artefacts whose truth is mathematically assured.

We create specifications for source code written in *C#* with corresponding specifications in its sister language called *Code Contracts (CodeCon)*. These act as a testbed for evaluating our bipartite creativity system. The core of this project is built on a general-purpose abduction system for discovering and extending similarities between general purpose knowledge graphs, being easily adapted for natural language and other data.

Extracting Code Graphs. The first process generates code graphs from the source code (Pitu, et al., 2013)[2], using 18 categories of nodes (*Variable, If, Block, Assign etc.*) connected by 6 types of relations (*Contains, Parameter, Returns*

[1] Code Contracts were chosen for this project as it supports all .Net languages (C++, Python, Java *etc.*), it estimates the completeness of a partial proofs and works in an ecosystem that may additionally support co-creativity for related software artefacts including test cases.

[2] https://www.kaggle.com/diarmuidodonoghue/graphs-of-code

etc.) between nodes. Each method is described by its own code graph, whose nodes are extracted from the source code. Edges represent relationships between nodes, depicting static relationships extracted from the abstract syntax tree generated by the compiler.

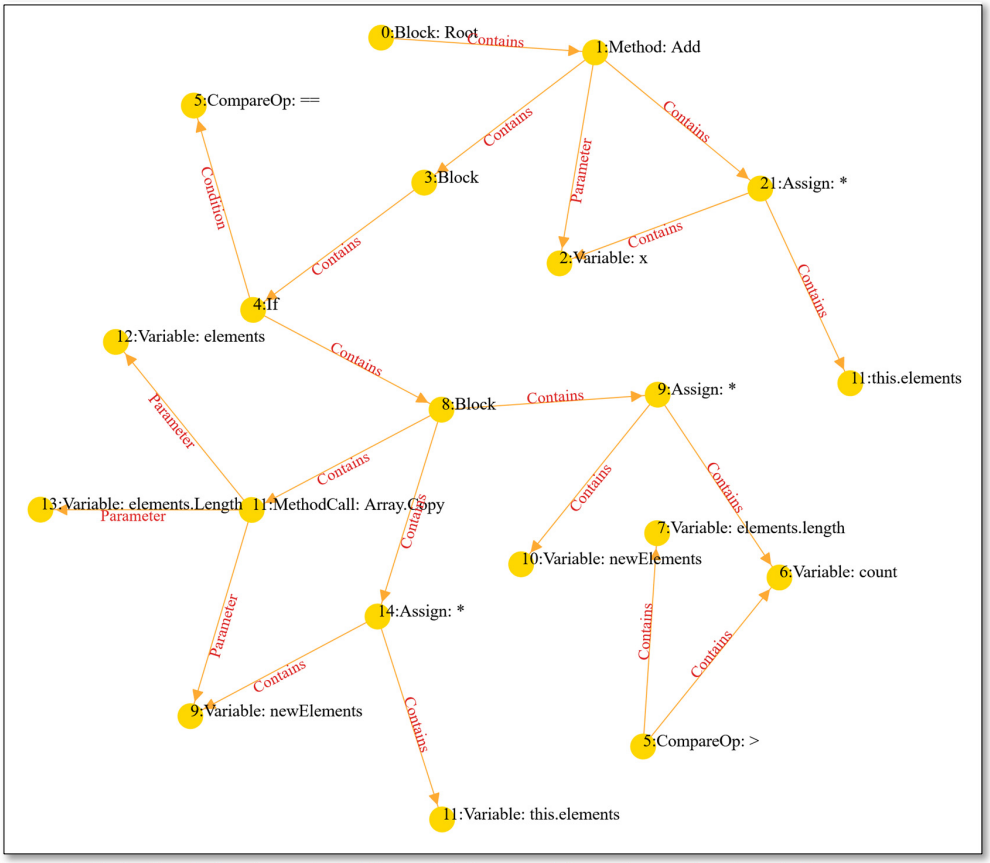

Figure 2: Code Graph for the code in Figure 1 is best read from the top-level node "`0: Block: Root`"

Finding Homomorphic Graphs. Abduction from a solution to problem code requires perceiving some reasonable level of similarity between code graphs. We identify functionally similar methods by a process of homomorphic matching (Carletti, 2020) between code graphs, using the NetworkX library and initial experiments using its ISMAGS algorithms. Later experiments use a combination of topological similarity while also exploiting the identical labels between paired nodes (so a `Block` node should match a `Block` node in the other graph). Subsequent phases of Aris exploit mappings containing paired `Variable` nodes in the two graphs, as these represent potential destinations for adding new specifications.

Infer and Adapt. The next phases translates CodeCon from the source into the problem graph, after first checking the compatibility of the proposed inference. This requires identifying the relevant C# source code to find the corresponding variable type as this isn't contained in the code graph. For example, if the CodeCon specifies `variable>=0`, this may be applied to both integer and real data types. This phase also uses the mapping to ensure the created specification interlocks with its new problem context. This phase also locates the correct point within the code to insert the CodeCon as this is essential for successful verification. This process uses the mapping and the source code of both methods to find the closest possible match to the original inspiring artefact.

Verify. Finally, the newly created code containing C# and CodeCon is added to its project for compilation, which automatically includes verification of any embedded CodeCon. Finally, all outputs are assessed to identify methods that were accepted by the deductive verifier.

Results and Discussion

We generated resulting using an inspiring set containing just 5 formally verified methods retrieved from the educational Rise4Fun website. Aris generated the code graphs and

iteratively attempted to abduce these specifications to a problem set containing around 1 million methods. While the ultimate objective is to create verified CodeCon, it is interesting to see how far along the Aris workflow each of the problems progressed as this gives us an indication of how like our process is to successfully extend to other source. Tables 1 and 2 quantify the number of candidate solutions as they progressed along the Aris workflow. "Potential targets" indicates the subset of methods selected for graph matching with that source. "Quality mapping" indicates the number of graph-matches above a hand-coded similarity threshold. The final two columns show the number of matches involving paired variable – a requirement for creating new specifications.

	Potential Targets	Quality mapping	1 mapped Variable	>1 Mapped variable
MaxDemo	145,824	98,323	62,189	12,468
ResizeDemo	11,142	9,845	5258	4,485
ResizeDemo Add	228,942	153,566	76,362	17,941

Table 1: Early Workflow Results

"Variable Type Match" in Table 2 indicates mapped variable have the same data type indicating a potential target for that specification. "Adapted" indicates that a new specification was created after adapting the inferred specification to better fit its new problem context.

			Successful Verification	
Variable Type Match	Adapted	Potentially Verifiable	# Code Con	Code Contr Density
489	2,111	489	5	52%
786	6987	786	3	to
2154	72,688	2,154	3	55%
		3,429	*11*	

Table 2: Late Workflow Results

"Potentially Verifiable" indicates specifications that we expect could successfully verify if we were able to reconstruct the entire project. Unfortunately, we were unable to verify many specifications because of incomplete repositories, code being incompatibility with the compiler version required by CodeCon and unavailable libraries.

The final two columns indicate successfully verified CodeCon. Aris used three specifications to create over 3,400 potentially verifiable new specifications, 11 of which were successfully verified. "Contract Density" is the deductive verifiers assessment of the completeness of that verification being slightly above 50%, indicating additional CodeCon are required for complete (100%) verification and an assurance that this source code can never yield an unexpected result.

Figure 2 depicts the typical distribution in similarity between a chosen graph and candidate graphs from the corpus. The vertical axis indicates the percentage of edges from the specified code graph that have been matched with a potential problem graph. The horizontal axis lists the similarity scores for a selection of similarly sized graphs, which here have been sorted according to that similarity score. The minimum level of similarity required to support creative abduction is an open question.

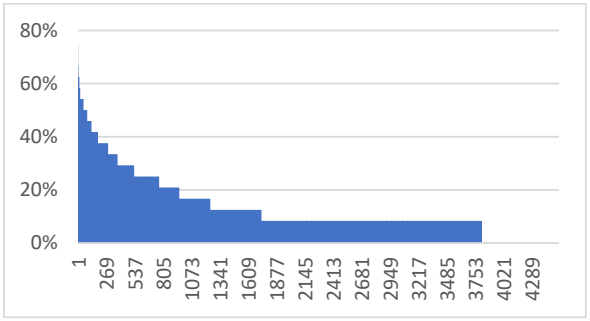

Figure 2: Exponential distribution of graph-based similarity estimates for one source

Static graphs. Our results are even more surprising as the static code graphs do not contain information on the relative ordering of statements within each *Block* – that is, there is no information indicating which statements occur first, second *etc*.

Self-Driving Creativity

Next, we see how a newly created artefact listed in Table 2 served to increase the creative ability of Aris. Because abductive creators can pinpoint the source of their creative leaps, we now show how a created artefact was used to drive subsequent creative inference. A graph was generated for the newly verified code and compared to the corpus. Figure 3 shows a problem method and highlighted (in yellow and boldface) is a newly created specification. Again, this method was successfully verified against the specification.

```
public static IEngineConfiguration-
TypeBuilder<TPoco> Value<TPoco, TMem-
ber>(this IEngineConfigurationTypeMember-
Builder<TPoco, TMember> memberConfig,
TMember value) {
  Contract.Requires(0 <= value);
  return memberConfig.Use<Val-
ueSource<TMember>> (new object[]
  { Value });
  }  }
```

Figure 3: The CodeCon in this code was created by Aris, using one of its own created artefacts.

Due to Aris' use of a graph matching process, the created artefact may match artefacts that did not match the initial artefact. Thus, a created artefact may increase the creative abilities of this abduction-based model. In (O'Donoghue, et

al., 2014) this is referred to as *self-sustaining creativity* where a created artefact serves to extend the creative potential of the system beyond the originally perceived limits. Figure 3 shows a "generic method" that was matched with a previously created specification.

The extreme lack of available specifications negatively impacts the diversity of the validated CodeCon. This lack of specifications can be addressed by a) manually creating more specification or, b) by looking to other languages as a possible source of specifications.

Between C# and Java

To find more formal specifications we next look to programs written in Java where the corresponding specifications are in its OpenJML sister-language. A corpus of around 10,000 code graphs was generated from open-source Java repositories. Figure 4 shows a Java method identified as similar to that of Figure 3 above, with this Java code clearly having much additional information. It is hoped we can explore bi-directional comparisons between languages, allowing transfer of specifications between them. We expect this approach to offer some additional CodeCon, however differences in coding language, specification language and underlying theorem provers may impinge upon the verification of some specifications.

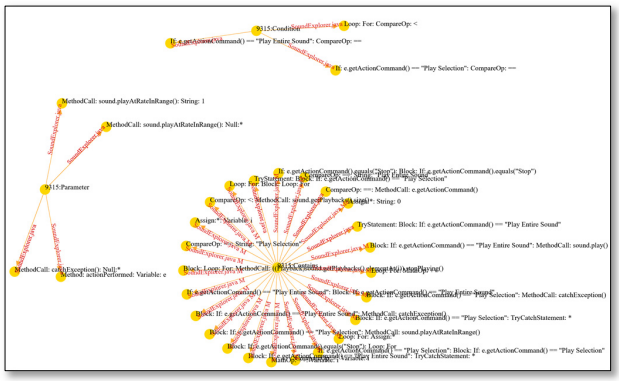

Figure 4: A matching code graph from Java

Future Work

We expect that our current results may be improved by improving graph mapping process and placing a greater focus on mappings between variables, as this may lead to generating more usable specifications. The Aris project may also allow us explore interactions between newly created artifacts and the verifier system (the Z3 SMT solver), because it can sometimes discover additional CodeCon that may help a human (or artificial) user to increase the CodeCon density of the solution. CodeCon can also generate a suite of test cases(data) using the PEX tool, further extending the range of items created from Aris specifications. However, there is currently no transferable learning between its creative episodes.

Conclusion

We describe a combination of abductive and deductive reasoning that suggests and then formally verifies new formal specifications for some given problem implementations. Aris detects similarity between static knowledge graphs derived from source code and creates formal specifications that were successfully verified. Furthermore, specifications created by Aris served to drive subsequent computationally creative episodes. The fundamental limits to self-sustaining creativity in this context remain to be explored.

References

Boden, M. (1992). *The Creative Mind.* Abacus.

Carletti, V. P. (2020). Comparing performance of graph matching algorithms on huge graphs. *Pattern Recognition Letters, 134*, 58-67.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Ponde, H., Kaplan, J., ... and Ray, A. (2021). *Evaluating Large Language Models Trained on Code.* arXiv preprint arXiv:2107.03374.

Colton, S. (2012). Automated theory formation in pure mathematics. . In *Springer Science & Business Media.*

Dross, Furia, Huisman, Monahan, & Müller. (2021). VerifyThis 2019: a program verification competition. *International Journal on Software Tools for Technology Transfer*, 1-11.

Huisman, M., Gurov, D., & Malkis, A. (2020). Formal Methods: From Academia to Industrial Practice. A Travel Guide. *arXiv preprint arXiv:2002.07279*.

Kaufman, J. C., & Beghetto, R. A. (2009). Beyond big and little: The four c model of creativity. *Review of general psychology, 13*(1), 1-12.

O'Donoghue, D., Monahan, R., Grijincu, D., Pitu, M., Halim, F., Rahman, F., . . . Hurley, D. (2014). Creating Formal Specifications with Analogical Reasoning. In *PICS - Publications of the Institute of Cognitive Science.*

O'Donoghue, D., Saggion, H., Dong, F., Hurley, D., Abgaz, Y., Zheng, X., . . . Zhao, X. (2014). Towards Dr Inventor: A Tool for Promoting Scientific Creativity. *International Conference on Computational Creativity (ICCC).* Slovenia.

O'Donoghue, D., & Keane, M. (2012). A Creative Analogy Machine: Results and Challenges. *International Conference on Computational Creativity (ICCC).* Ireland.

O'Donoghue, D., Power, O'Briain, Dong, Mooney, Hurley, . . . Markham. (2014). Can a Computationally Creative System Create Itself? Creative Artefacts and Creative Processes. *International Conference on Computational Creativity (ICCC).* Slovenia.

Pitu, M., Grijincu, D., Li, P., Saleem, A., Monahan, R., & O'Donghue, D. P. (2013). Arís: Analogical Reasoning for reuse of Implementation & Specification. *4th Artificial Intelligence for Formal Methods Workshop (AI4FM).* France.