Creating Formal Specifications with Analogical Reasoning

D.P. O'Donoghue[1] and **R. Monahan**[1] and **D. Grijincu**[1] and **M. Pitu**[1] and **F. Halim**[1] and **F. Rahman**[1] and **Y. Abgaz**[1] and **D. Hurley**[1]

Abstract. We describe the *Arís* (Analogical Reasoning for Implementations and Specifications) system that uses analogical reasoning to create formal specifications for a given implementation. Arís is built on the hypothesis that structurally similar implementations often represent similar functionality. It leverages this similarity to create new specifications, by analogy to a retrieved similar example. Of course some similarly structured implementations provide different functionality, so a major focus of Arís is to discriminate between analogous and dis-analogous pairs of code. Examples are used to highlight Arís' ability to create specifications, across a range of similar implementations and even similar algorithms. Results are presented on Arís ability to create verified specifications for a sample of ten textbook problems. We argue that Arís both emulates and supports the workaday *little-c creativity* of formal software developers**.**

1 INTRODUCTION

The *Arís*[2] system described in this paper, targets an audience of software developers through the creative reuse of existing formal specifications by re-applying them to new and dissimilar source code implementations. Writing computer source code is a complex and creative task [1], programmers exhibit Gardner's *little-c creativity* [2] as part of their regular production of new or updated software artefacts. But the task of writing formal specifications for source code can be even more challenging, requiring knowledge of the source code, the formal specification system and the underlying theorem prover being used to verify the correctness of the specifications' implementation. The system presented in this paper can be considered at Boden's [3] *combinatorial* level of creativity, combining the facilities of the Spec# and C# languages. However, one might consider each implementation and specification pair to be H-creative [3] artefacts. This paper uses an *inspiring set* of source code implementations (written in C#), for which we require accompanying specification code (in Spec#).

This paper adopts a process centred approach to creativity, based on the analogical reasoning process - for a wider discussion on process *vs*. product centred creativity and created artefact that are themselves processes see [4]. Analogical thinking is a very powerful technique contributing to the creativity of scientific, artistic and other disciplines [3, 4, 5, 6]. An analogy creates a new likeness between some problem (*target*) and some well-known *base* that is used to bring a new understanding to bear on that target. It has also been noted that creative insight often involves analogies with a strong imagery component [7]. Analogy models have previously addressed technical problems in; diagrammatic sketch understanding [8], processing topographic maps [9] and design [10].

We describe the *Arís* [11, 12] system that uses analogical reasoning to create new specifications for a given implementation, reusing similar – and not so similar - source code that already contains specifications. Arís [11] originally required near identical implementations to generate new specifications. But more recently [12] we have focused on creating specifications for less similar implementations. Arís adapts Gentner's Structure Mapping Theory [13] to the task of re-purposing formal software specifications from one implementation, for re-use in another implementation. This paper will make it clear that the Arís system possesses the essential qualities of creative systems: *novelty* and *quality* [3]**.**

The creation of new software artefacts has not received a great deal of attention from the computational creativity community. Disciplines like evolutionary and genetic programming routinely generate programs, but have not explicitly looked at producing creative outputs. Togelius *et al* [14] review the emerging topic of Procedural Content Generation devoted to the creation of new content for playable computer games. Cook *et al* [1] discuss the *MechanicMiner* system that generates new game mechanics for platform games, using evolutionary computing approaches. However, these systems are focused on creating new game dynamics and not on creating "general propose" software artefacts. *Rebuilder* [15] does address general purpose software design using analogical reasoning. However, Rebuilder is focused on software design specified at the UML class diagram level, whereas Arís focuses on the lower level of implementation and specification code.

The paper is structured as follows. First we describe the background to our work. We then explain how the Arís system identifies similar source code artefacts, re-purposing their specifications to create "similar" specification in the problem code. A simple example is used to highlight the operation of each phase of Arís. We then discuss a series of examples showing how Arís creates specifications for surprisingly different pairs of code. Results are presented of Arís creating specifications for 10 problem methods using source code information *only,* using a database of 43,051 methods.

2 BACKGROUND

Programming is often seen as a creative endeavour. A study of over 680 programmers involved in free/open source software [16] found that the most pervasive factor motivating their participation was *"how creative a person feels when working on the project"*. The authors note that, for example, writing a device driver for an

[1] Department of Computer Science, National University of Ireland Maynooth, Co. Kildare, Ireland. email: Diarmuid.ODonoghue@nuim.ie

[2] *Arís* is an Irish word meaning 'again' and is pronounced "ah-reesh".

operating system may not be considered very creative by an outside observer, it is often rated as very creative by those engaged in the project. They point to the subjective, personal interpretation of creative acts offered by Amiable [17]. The majority of respondents reported frequently losing track of time when engaged in programming – something that Amiable attributes to the "flow" [18] associated with creativity. Thus, we argue that producing source code is considered by many of those involved at least, to be a creative endeavour. We further argue that the production of formal specifications for these implementations can also be considered a creative endeavour.

Writing formal specifications for source code is frequently taught at advanced undergraduate or postgraduate levels. The objective is to write formal declarative "contracts" which describe the behaviour of a piece of software [19]. Automatic verification tools, such as the Spec# programming system [20], KeY [21] and ESC/Java [22] are used to statically verify that all future executions of the corresponding software are correct with respect to its contract.

Software verification is often associated with safety-critical systems (nuclear reactor control, air travel, telecommunications and transportation systems) but are also being deployed to other areas as the supporting tools become more advanced [23, 24, 25, 26]. Automatic verification tools are run like a compiler, returning a list of compilation/verification error messages, if they exist. These tools translate both the specification and the implementation to an intermediate representation language, from which proof obligations are generated using Dijkstras' weakest precondition calculus [27]. These conditions are input into various theorem provers/SMT solvers which discharge the proof obligations if the implementation is correct. If the implementation cannot be verified, error messages regarding the proof obligations that cannot be discharged are reported. Errors primarily arise due to either incorrect specifications or incorrect implementations. In these cases the user must revisit these program components to correct them. Errors can also occur due to the system's inability to automatically prove that the verification conditions can be discharged. In this case the user must interact with the verification system, by adding proof assisting assertions which contribute to the verification process.

In the past decade automated verification tools have become more powerful, primarily due to the major advances in SMT solver techniques. The verification tool used by Arís is the Spec# programming system [20]. Spec# is an automated verifier for C# programs, which uses the Boogie [28] intermediate representation language to help generate proof obligations, and a back-end SMT solver called Z3 [29] to discharge these obligations. While these tools are becoming more powerful, the major difficulties facing their users concern: (a) learning how to write good assertions to express what the program must achieve; (b) given a correct specification, produce a correct program whose verification is easily achieved; and (c) reuse proof strategies within the tools.

This paper discusses the Arís analogy-based system, which addresses these difficulties through offering computational creativity to an audience of formal software developers. It automates the steps involved in writing specifications, taking its inspiration from existing verified programs. For a given implementation without a formal specification, Arís retrieves similar verified code and seeks inspiration from the retrieved specification that accompanies the retrieved code. It both creates new formal specifications and assists developers in the process of creating computer code. Arís finds appropriate analogs for a presented problem and exploits these richer bases to suggest novel and useful inferences – in the form of Spec# code. The motivations for undertaking this work include: guaranteeing that software functions correctly, according to a given specification; reduction of the cost of developing formal software systems; and assisting developers in the creation of formal specifications. This is particularly beneficial when training software developers in a new discipline or when automatically generating formal specifications for libraries of code, so that it can be verified correct, to meet certification requirements. In addition, Arís explores the creative challenges found in a highly technical discipline, displaying some creative *flexibility* [30] and even greater *fluency*.

2.1 Analogy Models and Creativity

This subsection briefly reviews some previous models of analogical reasoning. MAC/FAC (Many are Called/Few Are Chosen) [31] models similarity based analogy retrieval, by a two-phase retrieval and mapping strategy. ARCS (Analog Retrieval by Constraint Satisfaction) [32] was also designed to identify the best analogy between a given target and a collection of base domains, incorporating semantic and structural factors in the process. Baydin *et al* [33] create analogs to describe a given problem, but do not look at inferences to extend that problem description. O'Donoghue *et al* [34] identified creative scientific analogies from within a corpus of domain descriptions, but this too relied on handcrafted data. Dr Inventor [35] aims to support creative reasoning for different types of automatically processed "research objects" [36], whereas Arís is specifically tailored to re-adapting existing software artefacts.

3 THE ARÍS SYSTEM

Arís (Analogical Reasoning for Implementations and Specifications) takes a problem method as its input and identifies functionally similar methods from its repository of methods. Arís is built on a multi-phase model of analogical reasoning encompassing the phases of; *representation, retrieval, mapping* and *validation*. By finding analogous source code, Arís aims to adapt the retrieved specification to its new problem context. We may also think of Arís as a Case-Based Reasoning (CBR) [37] assistant for software development, encompassing the phases of *retrieve, reuse, revise* and *retain*. We note other CBR systems have also been used to address creative challenges, from creative explanations [38] to creative design [15] .

Arís is an analogy-based system for creating formal specifications (written in the Spec# language) for a given implementation written in the C# programming language. Thus Arís creates new *precondition* (`requires`) statements, *post condition* (`ensures`) statements and *invariant* (`invariant`) statements that form verifiable formal contracts for the given source code. It also outputs *assume* and *assert* statements that act like directives to the underlying SMT solver, to overcome efficiency and proof strategy related issues. Like [31], Arís uses an inexpensive retrieval phase to identify candidate bases, which are then examined in more detail by the more detailed and expensive mapping phase.

3.1 The Code Base - Data

For this paper, we created a database of 43,051 C# programs obtained from SourceForge, CodePlex and similar repositories. Each program contained one method which typically consisted of approximately 30 lines of source code. However, only 127 Spec# specifications were contained in this database, obtained from only 29 methods. This small amount of specification code necessitated the creative re-use of existing specifications to as-broad-a-range of implementations as possible. This involved a flexible but still *little-c* creative process, creating and suggesting solutions for diverse pairs of source code artefacts.

We envisage Arís operating as a practical tool, creating specifications for a presented method. While Arís could iterate through all available methods to find a related specification, this "exhaustive search" approach would not scale well to large numbers. Instead we evaluate Arís using a more realistic scenario, retrieving relevant specifications based *only* on implementation details. Thus 43,022 of the available methods acted as "distractors", testing the ability of Arís to identify relevant specifications using only the implementation details.

```
public static int Summation(int k){
   int s = 0;
   for (int n=0; n < k; n++)
      s = s + n;
   return s;
  }
```

Figure 1: The target problem needing formal specification

The specifications that are required to enable the source code implementation in Figure 1 to be successfully verified (after variable renaming) are highlighted in the functionally similar source code in Figure 3. We highlight that "functionally similar" [39] and matching code may use different data types, different loop constructs, may involve additional unmatched identifiers and other differences. Arís was designed to support code reuse across this breadth of implementation details.

3.2 Representation

The fundamental unit of representation in Arís is a method of source code and the statements that implement that methods functionality. That is, each source and target analog represents the source code implementation of some method. In this paper we will focus on target problems that describe some implementation for which we require a formal specification. To this end, the source domains at our disposal contain both the implementations and associated specifications.

To support later processing the source-code is first translated into the abstract syntax tree by a compiler. This syntax tree is then processed to generate the corresponding *code graph*, which is a semantic network representation of the source code constructs and their inter-relationships. It is this code graph that is used by Arís for most of its activities. Code graphs use 18 categories of concept node (boxes with squared corners) in Figure 2, using the categories: *assignment, block, class, compareOp, Enum, field, if, logicalOp, loop, mathOp, method, methodCall, namespace, null, string, switch, tryCatch* and *variable*. There are 6 types of relation node (round-cornered boxes with italic text) in Figure 2, using the categories: *condition, contains, defines, depends, parameter and returns.* For example, a *loop* node can contain any of the following statements: `for`, `while`, `do-while` and `forEach` while a comparator node can contain: ==, >, <= *etc*.

A 30 line method typically produced a code graph containing around 78 concept and relation nodes. Most of Arís' subsequent processing uses code graphs rather than the "raw" source code. Multiply referenced variables such as input parameters are frequently involved in multiple relation nodes within a CodeGraph. In Figure 2 the input parameter "k" is also involved in a "CompareOp" node. Arís' analogy process will try to reuse specifications attached to similarly structured CodeGraphs.

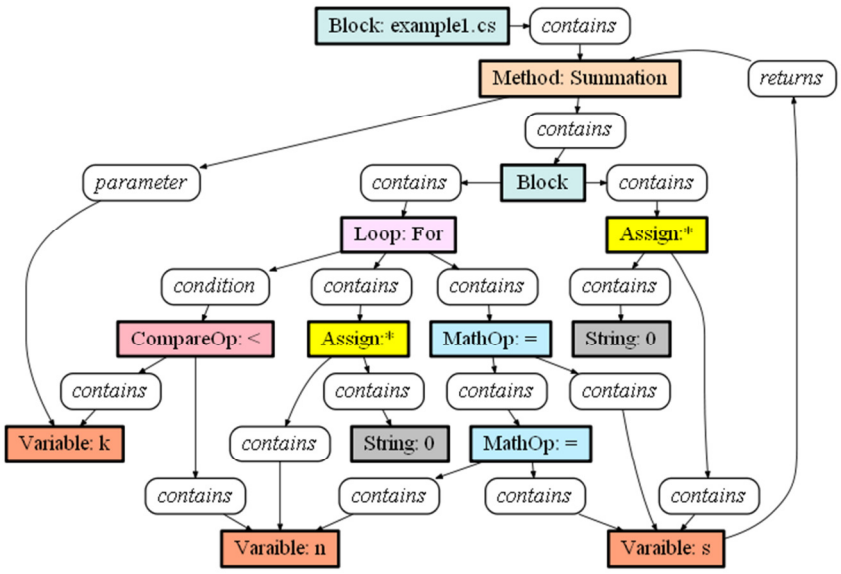

Figure 2: A *CodeGraph* depicting the deep structure of the source code listed in Figure 1

3.3 Retrieval

Rebuilder [15] has highlighted the crucial role that retrieval plays in the reuse and adaptation of pre-existing software artefacts. Like Rebuilder, Arís combines semantics and structure in its retrieval process to identify both isomorphic and homomorphic *code graphs*. Arís derives a vector space model from each code graph to allow quick and inexpensive comparison between vectors. Among the vector representations used are firstly, numeric identifiers for each distinct API method call. A document-term relevance (using TF-IDF) indicates the strength of the association between the API terms and pre-stored documents in the collection. Secondly, graph-based metrics derived from the code graphs, represent the number of concepts for each node type, number of relations, node references etc. Finally, Damerau-Levenshtein edit distance estimates the difference in the sequential arrangements of the API calls between methods. These metrics are generally very efficient at identifying a small fraction of the available bases that will be examined in greater detail by the subsequent mapping phase. A simple threshold is used to determine which items are deemed to be retrieved and which are not returned by the retrieval process.

3.4 Mapping

Arís next identifies the mapping between the base and target – to identify the detailed pre-existing similarities. Hard constraints upon the potential mappings considered ensure that only concept and relation nodes from the same categories can be mapped together. This greatly simplifies and expedites the mapping process, which is a variant on the NP-complete Maximum Common Sub-graph Isomorphism problem.

Arís is based on an incremental model of the mapping process [12, 34] that favours the development of mappings that place sequentially related lines of code in a mapping. It has a strong preference for isomorphic or near isomorphic mappings – within the bounds of the hard mapping constraint (a `for` node mapped to a `while` node). Arís uses a 2-way mapping between code graphs – by averaging the mapping size from base to target and also from target to base. This helps ensure the greatest similarity between the two code graphs. This was necessary because unmapped base or target code has a detrimental effect on the creation of new specification code. Use of a 2-way mapping appears to effectively overcome these problems. A mapping threshold rejects mappings that are too small to support viable inferences.

This mapping phase identifies paired items between the two implementations in Figures 1 and 3. As part of this it maps the `for` loop to the `while` loop, aligning their counter variables by sourcing them from the code graphs. In this way surprisingly dissimilar methods can form useful mappings.

Some of the items paired by Arís for the code in Figures 1 and 3 include: `Sum` <-> `summation`, `x` <-> `k`, `s` <-> `add`, `k` <-> `n`. However Arís is tolerant to the many differences within a mapping, including: `for` <-> `while` (different statements), `s=s+n` <-> `add+=k` (different expressions) as well as unmapped variables (`irrelevantVariable` in the base) and unmapped statements (`console.WriteLine` in the target). We note that all these differences occur within a single mapping between two code graphs, highlighting the wide range of differences that are allowed between two mapping fragments of code in Arís.

```
public static int Sum(int x)
 requires 0 <= x;
 ensures result==sum{int i in (0:x); i};
 {
  int add = 0; int k = 0;
  int irrelevantVariable = 0;
  Console.WriteLine(irrelevantVariable);
  while (k < x)
  invariant k <= x;
  invariant add == sum{int i in (0:k); i};{
      add += k; k = k-1; k = k +2;}
   return (int)add;
}
```

Figure 3: Arís identified this code as being analogous to the problem listed in Figure 1. Arís adapts the specifications (highlighted in yellow) to the problem code.

We also highlight some of the potential problems that may arise, even when very simple base and target constructs are mapped together. A base containing `while <condition>` might map with the target `while (true)`. But no loop invariants should be created for `while (true)` as the loop will never terminate (unless of course a `break` statement has also been used!). Such intricate problems pervade Arís' attempts to re-create specifications for non-identical methods. Distinguishing between analogous and *dis*-analogous bases is primarily addressed by the evaluation phase of Arís.

3.5 Inference and Validation

While the mapping process identifies pre-existing similarities, it is the generation of inference that creates new specification code. Arís uses the standard "pattern completion" algorithm [40] applied to the inter-code mapping, to create new Spec# code.

Not only does Arís support the creation of new formal specifications for a given segment of code, it also interacts with an external SMT solver to help ensure the quality of the C# implementation using those newly generated Spec# specifications. Spec# code that successfully verifies is automatically accepted. However, unverified Spec# code is presented to the (human) user for possible adaptation, thereby using Arís in its role as a creativity assistant tool.

Not only must Arís generate the correct specifications, but must also insert them in the correct location in the problem code. The first two specification components (the precondition and the post condition) must be placed between the method header and the opening bracket. The two `invariants` (which state the conditions that should be true every time the loop iterates) must be placed immediately before the opening bracket of the `for` loop. We point out that variables `k` and `add`, in Figure 3, are referenced both by the implementation and the specification, allowing the tools to reason about the correctness of the specification as realised by the given implementation.

4 ARÍS AS A CREATIVITY TOOL

In the simplest case Arís might retrieve source code that is lexically identical to a presented problem. In such a situation involving two lexically identical implementations, the specifications can be

trivially transferred between the two methods. However, presenting an implementation that is lexically identical (sometimes called code clones) to some stored code is extremely infrequent.

Even very small differences between two implementations can cause significant differences in behaviour, making any pre-existing specifications in one inapplicable to the other implementation. Despite such problems, Arís is surprisingly successful at creating new specifications that automatically verify using the Spec# programming system. For the *summation* example in Figures 1 and 3 above, Arís identifies the correct mapping and the newly created specifications verify automatically with the given target problem.

We argue that because Arís can produce new specifications for target problems that are surprisingly different from the available bases, its outputs (source code plus specifications) can be considered *novel*. The argument that its outputs are *useful* would appear to be more obvious. The argument in support of the creativity of Arís may be supported using Gardner's four facets of creativity (in [2] pp 33-35). Firstly, Arís is aimed at practitioners working within a specific domain – in this case formal software developers (which Gardner contrasts with general purpose creativity). Secondly, Arís is aimed at being regularly creative (at different levels), rather than being a tool for 1-off creativity. However, Arís does not possesses the ability to *"ask new questions"* of a domain. Finally, the acceptance by a culture of its creative outputs is crucial to Arís – in terms of automatic verification of its outputs (using a theorem prover) and in its role as a "creative assistant" to prompt a software engineer with potentially useful but unverified specifications.

5 ARÍS AS A CREATIVITY ASSISTANT

While similar implementations can sometimes lead to the reuse of associated specifications, there is no guarantee that these specifications will be valid within the target context. Arís also creates many more specifications that are rejected by a verification tool but that might prompt a novice specification writer to create a verifiable specification.

5.1 Analogous Implementations

Even very small differences between mapped portions of source code can impact the validity of associated specifications. The most similar implementation found in a repository to a given problem will often contain many of these differences, when aggregated together may not support the same formal specifications. However, some dissimilar implementations will inevitably be functionally similar and should support the same types of formal specification. Adapting specifications between similar (but *not* identical) implementations is a creative endeavour.

Even a single character difference between the base and problem can result in the newly created specifications being rejected at the verification stage. For example, a base program that calculates the minimum of two variables *x* and *y* will have a post condition of the form `ensures (x<y)? result == x: result == y`. Now consider a target that calculates the maximum of two variables `a` and `b`. Arís will match with this program and a new specification will be created, renaming the variables to *a* and *b*. However, the minimum program with the post condition above will not be automatically verified by the Spec# tools. However, the failed specification might prompt the user to change the "<" character and create the correct post condition `ensures (a>b)? result == a: result == b.`

A similar situation can be seen where one formally specified program sorts an array into increasing order and another program without a specification sorts an array into decreasing order. The specifications created by Arís for the second program, do not verify automatically. The theorem prover issues warning messages indicating an unsatisfied post condition and a loop invariant that does not hold. But using Arís as a creativity assistant, a software developer might again change the "<" (in the specification code below) to ">" for successful verification. Thus the rejected specification prompts the user to create the successful specification `-ensures (result == true) ==> forall{int i in (0 : arr.Length-1); arr[i] <= arr[i+1]}`

5.2 Analogous Algorithms

The next example involves mapping different algorithms as well as implementation details. When verifying a binary search program (which searches within sorted data), Arís may match with an implementation of a Linear Search algorithm (performing search in un-sorted data). In this case the post condition of the linear search can be re-applied to the Binary Search algorithm. The implementations are also significantly different where one simply searches through the sequence of unordered data until the value is found and the other uses a partitioning technique on the ordered data to find the value more efficiently. The post condition for both of these algorithms are identical but the users creativity is required for the precondition of the Binary Search, stating that the input data is ordered i.e. `forall{int i in (0: a.Length), int j in (i: a.Length); a[i] <= a[j]}`

5.3 Further Creative Proof Strategies

As already shown using loop invariants, verification tools often require the user to provide information in addition to preconditions and post conditions, in order to contribute to the proof strategy used to achieve automatic push button verification. This additional information takes the form of the loop `invariant`, which states the conditions that are true throughout an iterative section of code; `assert` statements, which indicate to the verifier which conditions it should check; and `assume` statements, which provide user-added assumptions to the proof strategy used in the verification. These invariant, assertions and assumptions are all added to the source code as annotations. Typically usage of `assert` is in verifying loop termination and in providing hints to the verifier to direct the overall proof strategy. Arís allows for the transfer of assertions to new program verifications, in both scenarios. While developing a new proof, assumptions such as `assume x != null` can be used to modularise the proofs, thus assisting the users creativity. Using Arís we reduce this proof strategy burden on the user by detecting such annotations in similar problems and presenting them for adaptation by the user so that they can be used in the verification of the new problem.

6 RESULTS OVERVIEW

This section provides an overview of the results produced by Arís. The source code for each ten methods (listed in Table 1) was presented to Arís, which then attempted to produce new specifications for each of these methods. The codebase described in Section 4.1 contained all 43,051 methods as well as a small number of implementations that also contained specifications. The methods discussed here are representative of the problems found in competitions such as VerifyThis.org and VSComp.org.

The first three columns of Table 1 details the results produced by the retrieval phase of Arís. On average 70% of the codebase was retrieved for each problem. Thus it appears that the retrieval phase was not particularly successful. However we point out that the absence of API calls in most of these problem methods means that these results are worse than generally achieved on "richer" problem code.

Table 1. Retrieval and Mapping performance of Arís.

	Quantity of methods			
Method name	□	■	○	●
Min(int, int)	39067	90.7	456	1.1
Max(int,int)	39067	90.7	455	1.1
Coincidence(int[],int[])	3769	8.8	362	0.8
BinarySearch1(int[],int)	25	0.1	353	0.8
LinearSearch(int[] a, int key)	39178	91.0	478	1.1
Sum(int[])	38804	90.1	409	1.0
Count(int[], int)	39244	91.2	469	1.1
ISqrt(int)	37817	87.8	386	0.9
Factorial(int)	38804	90.1	438	1.0
CountNonNull(string[])	38637	89.7	469	1.1
average	*31441*	*73.0*	*427.5*	*1.0*

□ Number of methods identified by the retrieval process ■ % of entire database retrieved ○ number of methods mapped (above the threshold) ● % of the entire code-base that mapped.

The last two columns of Table 1 summarise the results produced by the mapping phase of Arís. These columns detail the number of viable mappings produced for the ten target problems (i.e. the size of the mapping was above a pre-set threshold). This mapping process identified an average of just 1% of the available methods as similar to the presented target problems. This identified any code graphs that contained large systemic similarities to the target problem, encompassing isomorphic and homomorphic code graphs. Thus, the mapping phase of Arís appears to accurately discriminate between different code structures.

Table 2. The number of specification statements that were created (but not necessarily verified) by Arís.

	Quantity of		
Method name	requires	ensures	invariant
Min(int, int)	2	3	2
Max(int,int)	2	4	0
Coincidence(int[],int[])	8	8	13
BinarySearch1(int[],int)	11	10	16
LinearSearch(int[] a, int key)	6	4	12
Sum(int[])	6	5	14
Count(int[], int)	7	8	14
ISqrt(int)	3	5	3
Factorial(int)	4	4	9
CountNonNull(string[])	6	7	14
total	*55*	*58*	*97*

Table 2 summarises results produced by Arís inference and validation phases. This lists the number of new Spec# statements that were generated for each of the ten methods. As can be seen 210 specification statements were created, comprised of 55 `requires` (preconditions), 58 `ensures` (post-conditions) and 97 `invariant` statements. That so many Spec# statements were produced was seen as an excellent result and was pleasantly surprising. However, not all of these statements were successfully verified, but it was nevertheless a promising result.

Table 3. The number of specification statements created and verified by Arís.

	Quantity of specification types		
Method name	requires	ensures	invariant
Min(int, int)	0	2	0
Max(int,int)	0	2	0
Coincidence(int[],int[])	3	1	8
BinarySearch1(int[],int)	3	8	11
LinearSearch(int[] a, int key)	6	1	7
Sum(int[])	3	1	3
Count(int[], int)	3	2	6
ISqrt(int)	1	2	2
Factorial(int)	1	1	2
CountNonNull(string[])	4	2	9
total	*24*	*22*	*48*

Table 3 details the number of verified specification statements produced by Arís on this problem set. Arís created 24 new precondition (`requires`) statements, 22 post conditions (`ensures`) and 28 invariants. Thus for these 10 problem methods Arís created a total of 74 verified specification statements, each being successfully verified by an SMT theorem prover. Interestingly, Arís generated at least two specifications for each of these problems and even managed to generate twenty two specifications for one problem.

7 ARÍS AND OTHER LANGUAGES

Recent developments have focused on enabling Arís to interchange specifications with other implementation and specification language-pairs (e.g. Java and JML - Java Modeling Language [41]). The objective is to make Arís the central engine for many implementation languages, using extensions to create code graphs and to create specifications in new languages. The main architectural additions involve generating code graphs for a new implementation language and translating the new Spec# specifications into a new specification language.

Arís was presented with a Java implementation of the summation method in Figure 1. Using a code graph derived from Java code it identified the correct C# method and re-created the required specification – which Arís then translated from Spec# into the JML specification language. Extending Arís in this way supports creativity with a range of programming languages while reusing the core Arís engine. In addition to providing a framework for using Arís with new languages where the input and output is the same, our framework allows for creativity between languages e.g. while the input language could be Java, the generated implementation and specification could be expressed in the Eiffel language. This facilitates the creation of new problem solutions

which allow access to an extended range of tools and techniques for software verification.

```
  requires 0 <= x;
  ensures \result == (\sum int i;  0 <= i &&
i < x; i);
  public static int sum(int x){
    int add = 0;
    int k = 0;
    maintaining k <= x;
    maintaining add == (\sum int i;  0 <= i
&& i < k; i);
    //@ (* decreasing x - k *);
     while(k < x){
         add += k;
         k++; }
     return add; }
```

Figure 4: Arís created specifications (in yellow) in JML (Java Modelling Language) for a Java implementation using a C# and Spec# as its base

8 CONCLUSION

This paper presented the *Arís* analogy-based system that creates new formal specifications (in Spec#) for a presented implementation (written in C#). Arís is effective at retrieving functionally similar implementations to some presented problem code. It successfully generates new specifications for that code, based on the identified analogous structure between the paired source code constructs.

On a test using ten methods requiring specifications Arís' retrieval phase did not appear to be particularly accurate. However, its mapping phase accurately identified around 1% of the codebase as similar to each presented problem. Arís created 74 verified specification statements, composed of 24 precondition (`requires`) statements, 22 post conditions (`ensures`) and 28 invariants. In addition Arís also produced a further 31 unverified preconditions, 36 post conditions and 49 invariants. Many of these "failed" specifications might prompt the creativity of a human software developer to write successfully specifications.

Little-c creativity [2] can be seen at work in Arís' ability to both create new specifications and to assist in their creation. Examples highlighted Arís' ability to create new specifications for similar and surprisingly dissimilar implementations, even creating specifications for different algorithms. Arís may be seen to operate at Boden's *combinatorial* level of creativity. In the future Arís might even approach *exploratory* creativity, creating new proof strategies arising from non-obvious comparisons between implementations.

ACKNOWLEDGEMENTS

The research leading to these results has received funding from the European Union Seventh Framework Programme [FP7/2007-2013] under grant agreement 611383. This project has also been partly funded with support from the European Commission. This publication reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

REFERENCES

[1] M. Cook, S. Colton, A. Raad and J. Gow, "Mechanic Miner: Reflection-Driven Game Mechanic Discovery and Level Design," *LNCS Vol 7835,* pp. 284-293, 2013.

[2] H. Gardner, Creating Minds, Basic Books, NY., 1993.

[3] M. Boden, The Creative Mind, Abacus, 1992.

[4] D. O'Donoghue, J. Power, S. O'Briain, F. Dong, A. Mooney, D. Hurley, Y. Abgaz and C. Markham, "Can a Computationally Creative System Create Itself? Creative Artefacts and Creative Processes," in *International Conference on Computational Creativity*, Slovenia, 2014.

[5] A. Koestler, The Act of Creation, Penguin Books, NY, 1964.

[6] M. Boden, "Computer Models of Creativity," *AI Magazine,* pp. 23-34, 2009.

[7] J. Davies, A. K. Goel and P. W. Yaner, "Proteus: Visuospatial Analogy in Problem Solving," *Knowledge-Based Systems,* vol. 21, no. 7, 2008.

[8] K. Forbus, J. Usher, A. Lovett, K. Lockwood and J. Wetzel, "CogSketch: Sketch Understanding for Cognitive Science Research and for Education," *Topics in Cognitive Science,* pp. 1-19, 2011.

[9] D. P. O'Donoghue, A. Bohan and M. Keane, "Seeing Things: Inventive Reasoning with Geometric Analogies and Topographic Maps," *New Generation Computing,* vol. 24, no. 3, pp. 267-288, 2006.

[10] A. K. Goel and S. R. Bhatta, "Use of design patterns in analogy-based design," *Advanced Engineering Informatics,* vol. 18, no. 2, p. 85–94, 2004.

[11] R. Monahan and D. P. O'Donoghue, "Case Based Specifications – reusing specifications, programs and proofs," *Dagstuhl Reports,* vol. 2, no. 7, pp. 20-21, 2012.

[12] M. Pitu, D. Grijincu, P. Li, A. Saleem, R. Monahan and D. P. O'Donoghue, "Arís: Analogical Reasoning for reuse of Implementation & Specification," in *AI4FM - 4th International Workshop on Artificial Intelligence for Formal Methods*, Rennes, France, 2013.

[13] D. Gentner, "Structure-Mapping: A Theoretical Framework for Analogy," *Cognitive Science,* vol. 7, no. 2, pp. 155-170, 1983.

[14] J. Togelius, G. Yannakakis, K. Stanley and C. Browne, "Search-based Procedural Content Generation: A Taxonomy and Survey," *IEEE Transactions on Computational Intelligence and AI in Games,* vol. 3, no. 3, pp. 1-15, 2011.

[15] P. Gomes, N. Seco, F. Pereira, P. Paiva, P. Carreiro, J. Ferreira and C. Bento, "The importance of retrieval in creative design analogies," *Knowledge-Based Systems,* vol. 19, pp. 480-488, 2006.

[16] L. K.R and W. R.G., "Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects," in *Perspectives on Free and Open Source Software*, F. J., F. B., H. S. and a. L. K.R., Eds., MIT Press, 2005, pp. 1-27.

[17] T. Amabile, Creativity in context, Boulder, CO:: Westview Press, 1996.

[18] M. Csikszentmihalyi, Creativity: Flow and the Psychology of Discovery and Invention., New York.: Harper Collins, 1996.

[19] B. Meyer, Object-Oriented Software Construction, Prentice Hall, January 1997.

[20] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Mülller, W. Schulte and H. Venter, "Specification and Verification: The Spec# Experience," *CACM,* 2010.

[21] B. Beckert, R. Hahnle and P. H. Schmitt, "Verification of Object-Oriented Software: The KeY Approach," in *Springer-Verlag* , Berlin, Heidelberg, 2007.

[22] P. Chalin, J. Kiniry, G. T. Leavens and E. Poll, "Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2," in *The 4th Internaitonal Symposium of Formal Methods for Components and Objects (FMCO'05)*, Amsterdam, The Netherlands, November 2006.

[23] J. Woodcock, P. G. Larsen, J. Bicarregui and J. Fitzgerald, "Formal Methods: Practice and Experience," *ACM Computer Survey,* vol. 41, no. 4, p. Article 19, October 2009.

[24] C. Jones, P. O'Hearn and J. Woodcock, "Verified Software: A Grand Challenge," *Computer,* vol. 39, no. 4, pp. 93-95, April 2006.

[25] C. Jones and A. Romanovsky, "Special Issue on Automated Verification of Critical Systems (AVoCS'11)," *Science of Computer Programming,* vol. 82, no. 0, p. 1, March 2014.

[26] Y. Moy, E. Ledinot, H. Delseny, V. Wiels and B. Monate, "Testing or Formal Verification: DO-178C Alternatives and Industrial Experience," *IEEE Software,* vol. 30, no. 3, pp. 50-57, 2013.

[27] E. W. Dijkstra, A Discipline of Programming (1st ed.), Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997.

[28] M. Barnett, B. Chang, R. DeLine, B. Jacobs and K. Leino, "Boogie: A Modular Reusable Verifier for Object-Oriented Programs," *FMCO,* 2005.

[29] N. De Moura and L. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems - LNCS 4963*, 2008, pp. 337-340.

[30] J. P. Guilford, "Creativity," *American Psychologist,* vol. 5, no. 9, pp. 444-454, 1950.

[31] K. Forbus and D. Gentner, "MAC/FAC: A model of similarity-based retrieval," *Proceedings of the Cognitive Science Society,* 1991.

[32] P. Thagard, K. J. Holyoak, G. Nelson and D. Gochfeld, "Analog Retrieval by Constraint Satisfaction," *Artificial Intelligence,* 1990.

[33] A. G. Baydin, R. de Mantaras and S. Ontanon, "A semantic network-based evolutionary algorithm for modeling memetic evolution and creativity," *Neural and Evolutionary Computing,* 2014 (pending).

[34] D. O'Donoghue and M. Keane, "A Creative Analogy Machine: Results and Challenges," in *Proc. 4th International Conference on Computational Creativity (ICCC)*, Dublin, Ireland, 2012.

[35] D. O'Donoghue, H. Saggion, F. Dong, D. Hurley, Y. Abgaz, X. Zheng, O. Corcho, J. Zhang, J.-M. Careil, B. Mahdian and X. Zhao, "Towards Dr Inventor : A Tool for Promoting Scientific Creativity," in *International Conference on Computational Creativity*, Ljubljana, Slovenia.

[36] Ó. Corcho, G. V. Daniel, K. Belhajjame, J. Zhao, P. Missier, D. Newman, R. Palma and e. al, "Workflow-centric research objects: First class citizens in scholarly discourse," in *9th Extended Semantic Web Conference*, Hersonissos, Greece, 2012.

[37] D. M. R.L., D. McSherry, D. Bridge, D. Leake, B. Smyth, S. Craw, B. Faltings, M. Maher, M. Cox, K. Forbus, M. Keane, A. A and W. I., "Retrieval, reuse, revision and retention in case-based reasoning," *The Knowledge Engineering Review,* vol. 20, no. 3, pp. 215-240, 2006.

[38] D. Leake, "Abduction, experience, and goals: a model of everyday abductive explanation," *Journal of Experimental & Theoretical Artificial Intelligence,* vol. 7, no. 4, 1995.

[39] B. Dit, B. Revelle, M. Gethers and M. Poshyvanyk, "Feature Location in Source Code: A Taxonomy and Survey," *Journal of Software:Evolution and Process,* vol. 25, no. 1, pp. 53-95, 2013.

[40] K. J. Holyoak, L. Novick and E. Melz, "Component Processes in Analogical Transfer: Mapping, Pattern Completion and Adaptation," in *Analogy, Metaphor and Reminding*, 1994.

[41] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kuniry, G. T. Leavens, K. R. M. Leion and E. Poll, "An Overview of JML tools and applications," *International Journal on Software Tools for Technology Transfer,* vol. 7, no. 3, pp. 212-132, June 2005.