Arís[1]: Analogical Reasoning for reuse of Implementation & Specification.

Pitu M., Grijincu D., Li P., Saleem A., Monahan, R. O'Donoghue D.P.

Department of Computer Science, NUI Maynooth, Co. Kildare, Ireland.

Introduction: Formal methods and formal verification of source code has been used extensively in the past few years to create dependable software systems. However, although formal languages like Spec# or JML are quite popular, the set of verified implementations remains small. Our work aims to automate some of the steps involved in writing specifications and their implementations, by reusing existing verified programs i.e. for a given implementation, we aim to retrieve similar verified code and then reapply the missing specification that accompanies that code. Similarly, for a given specification, we aim to retrieve code with a similar specification and use its implementation to generate the missing implementation.

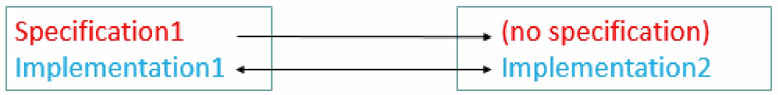
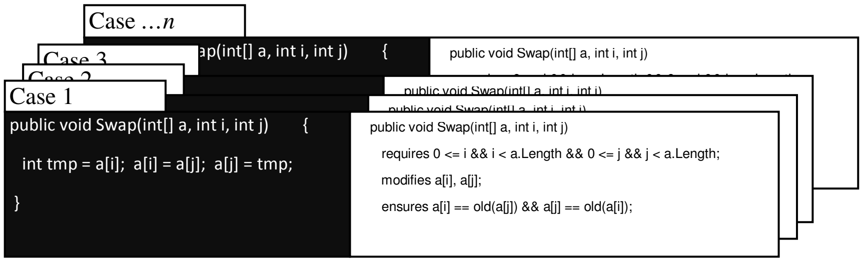


Figure 1: *Example: Similar implementations reusing specifications*

Reasoning by Analogical Comparison: One of the more successful disciplines in artificial intelligence in recent years has been Case-Based Reasoning (CBR). While CBR traditionally uses relatively straightforward "cases", retrieving similar implementations requires structure rich "cases" and necessitates CBR's parent discipline of Analogical Reasoning (AR). Both AR and CBR solve problems not from first principles, but by using old solutions to solve new problems. We use these problem solving disciplines to assist the reuse of verified programs.

The canonical analogical algorithm is composed of the phases: *Representation, Retrieval, Mapping, Validation* and *Induction*. This paper focuses on *Representation, Retrieval and Mapping*. Currently in *Arís*, our model for verified program reuse, we focus on code retrieval. *Representation* depicts problems (and solutions) as static parse trees. We then *retrieve* similar parse trees for a given problem. Next, we identify the *mapping* between the two parse trees in order to generate the analogical inferences – transferring and adapting the retrieved specification to the given problem. Our parallel work on specifications explores specification matching and specification reuse. We briefly discuss this work later.



```
public void Swap(int[] a, int i, int j)    {
  int tmp = a[i];  a[i] = a[j];  a[j] = tmp;
}
```

```
public void Swap(int[] a, int i, int j)
  requires 0 <= i && i < a.Length && 0 <= j && j < a.Length;
  modifies a[i], a[j];
  ensures a[i] == old(a[j]) && a[j] == old(a[i]);
```

Figure 2: *The Case-Base contains Source code implementations and corresponding formal specifications*

Code Retrieval: When presented with a problem implementation, *Arís* beings by retrieving a similar implementation so as to re-use its specification. Each "case" is a software artefact which varies in granularity from methods, to classes or full applications. Our current model is implemented to perform retrieval using C# source code and corresponding Spec# (Leino & Müller, 2009) specifications. Retrieval ranks code artefacts by their similarity score with a given input artefact.

[1] Meaning "again" in the Irish Language

(Mishne & De Rijke, 2004) successfully used *conceptual graphs* as a basis for source code retrieval. A *conceptual graph* (Sowa, 1994) is a bipartite, directed and finite graph, in which a node has an associated type (can be either a *concept node* or a *relation node*) and a referent value (the content inside the node). We use the concept nodes to model different structures and attributes from the source code (e.g. *Loop* concept) and relation nodes to show how they relate to one another (e.g. a concept node of type *Method* may have an ongoing edge to a relation node of type *Parameter*). The advantage of using this kind of representation is that it offers the possibility of exploring the semantic content of the source code while also analyzing its structural properties using graph-based techniques. In order to create the conceptual graph we first parsed the source code into an Abstract Syntax Tree (using the *Microsoft Roslyn* API) and then transformed the result into the higher level representation of a conceptual graph.

Source code retrieval is performed using distinct semantic and structural characteristics of the source code. In the semantic retrieval process, we express the meaning of the code through the use of API calls. API calls have been used as semantic anchors in (McMillan, Grechanik, & Poshyvanyk, 2012) as a successful method of retrieving similar software applications, because they have precisely defined semantics - unlike names of program variables, types and words that programmers use in comments (techniques used in the majority of existing source code retrieval systems). We further rely on the Vector Space Model (Salton, Wong, & Yang, 1975) to represent our documents (code artefacts) and use API calls as “words” in these documents to support semantic retrieval.

In the structural retrieval process, we focus on source code topology as represented in these conceptual graphs and by analysing metrics of these representations (for example, number of nodes, number of loops, loop size, connectivity (O'Donoghue & Crean, 2002)). Based on conceptual graphs, we extract content vectors (Gentner & Forbus, 1991) which express the structure of the code and the number of concepts (for example, loops, statements, variable declarations, etc.). Because our database of source code artefacts will potentially be very large, we need an efficient way of retrieving the most similar cases. We propose using the K-means clustering algorithm in order to create sub-groups of content vectors and perform K-nearest neighbours only on the “closest” sub-group to a given input content vector, thus speeding up the structural retrieval process. We then combine semantic retrieval with structural retrieval and impose different constraints: depending on the type (method, class, application) of the input artefact (query) - retrieving only artefacts of the same type. The outcome of retrieval then, is a ranked list with a small number of the *most* similar artefacts found in the repository.

Code Mapping: This next mapping task is a per-cursor to solution generation, finding detailed correspondences between the old solution and the new problem. We use the term *source* to refer to each retrieved (candidate) solution in turn and the term *target* to refer to our unspecified problem code. The objective here is to identify the best mapping between the two isomorphic graphs, where the two programs use different identifier names. But mapping should also cater for homomorphic graphs, allowing different structures to be mapped together. For code matching we propose to use an incremental matching algorithm based on the *Incremental Analogy Machine* (IAM) (Keane, Ledgeway, & Duff, 1994). Although methods for comparing conceptual graphs have been proposed before, many of them focus on matching identical graphs or subgraphs (like Sowa’s set of projections and morphisms) or they rely on various parameters that have to be empirically determined (Mishne & De Rijke used parameters like concept and relation weights, matching depth *etc*). IAM begins by matching the two largest sub-trees within the source and target graphs. This forms a *seed mapping* and then additional structures from the source and target are added iteratively forming a single mapping between the new code and the previously specified code.

Mapping Constraints: Specific constraints guide the formation of this mapping. Gentner’s (1982) 1-to-1 constraint ensures that the mapping remains consistent. We also impose further constraints on those concepts that are mapped between the source and target code. We might ensure for example that variables are matched with variables and loops with loops. Isomorphic subgroups that share the same structural properties are then formed based on one-to-one correspondences between the graph concepts. However, we plan to map graphs that don’t share the exact same structure, but that may be related (see for example the two graphs in Figure 3). We are currently exploring with mappings between two non-isomorphic (homomorphic) sub-graphs from the source and target parse trees.

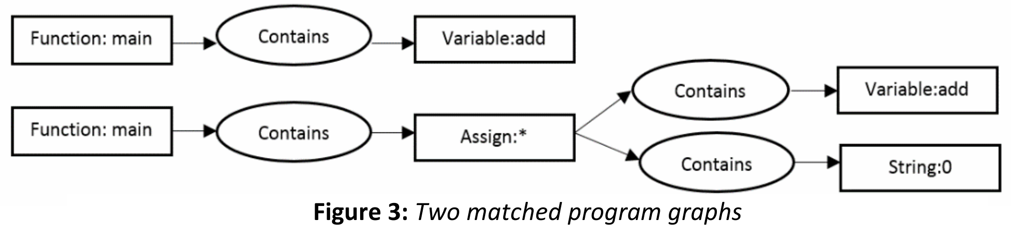


Figure 3: *Two matched program graphs*

Finding the appropriate "root" concept (the most referenced node in the graph) from which to start the matching process is a challenging aspect of using the IAM algorithm. We will address this issue by assigning node ranks (using a graph metric akin to Page Rank like the one proposed in (Bhattacharya, Iliofotou, Neamtiu, & Faloutsos, 2012)) and see whether this will lead us to conclusive mappings between the source and target domains. The generated mapping is then evaluated to decide whether or not it is near-optimal and the algorithm may choose to backtrack to select alternative seed mappings – or the mapping may be abandoned if sufficient similarity cannot be found.

Inference: Once IAM has found a suitable mapping, it generates the analogical inferences in order to generate the required specifications for the given code. Analogical inferences are generated using a surprisingly simple algorithm for pattern completion called CWSG - Copy With Substitution and Generation (Holyoak *et al*, 1994). CWSG transfers the additional specifications from the retrieved code and adds it to the target code – substituting source code items with the mapped equivalents. This should allow our target/problem code to be formally verified using the newly generated specification. Optionally, we can also retain the newly formally verified source code artefacts for further use.

Parallel Research on Specification Matching and Reuse: In addition to our work on code reuse we are exploring the reuse of specifications. For a given specification, we aim to retrieve code with a similar specification and use the retrieved implementation to generate the "missing" implementation. The work here has two approaches: the matching of specifications based on the description of the program's behaviour and the reuse of the same specifications for implementations that differ only in their use of data structure.

The first approach focuses on the specification matching of software components (Zaremski & Wing 1997) using a hierarchy of definitions for precondition and postcondition matching. We apply these definitions to specifications of C# code that are written in Spec# (Barnett et al. 2005) using the underlying static verifier Boogie (Leino et al. 2005) and the SMT solver Z3 (De Moura and Bjørner, 2008) to determine matches and to verify the correctness of our specification implementation pairs. Results to date are promising with a mixture of method specification matches allowing the retrieval and verification of similar specifications and their associated implementations.

Our second approach focuses on reuse of specifications and program verifications. Software clients should only be concerned with specifications and do not need to know details about the implementation and the verification process. This separation of concerns is achieved via data abstraction where we provide an abstract view of a program which we can provide to the client without exposing implementation details. As a result, each specification may have many implementations, each differing in terms of the underlying data structure used in their implementations. The theory of data refinement guarantees that this difference of data structure does not adversely affect the correctness of the programs with respect to their specifications. Our research here explores the reuse of specifications, and the automatic generation of the associated proof obligations, when an implementation is replaced by another implementation that is written in terms of an alternative data structure. We focus on the Dafny language (Leino, 2010), which is closely related to the Spec# language and is verified using the Boogie static verifier. The advantage of using Dafny is that it offers updatable ghost variables which can be used to verify the correctness of a data refinement.

Future work: The basic assumption underlying our work is that similar implementations have similar specifications. Further work is ongoing to ascertain the veracity of this hypothesis – and if true, what degrees of "similarity" are required. At the moment our work is fragmented by slightly different tools and approaches. In the future, we hope to combine these approaches within ***Arís*** to achieve a fully integrated platform with analogical reasoning as its core, allowing for reuse of implementations, specifications and their verifications.

References

Barnett M., DeLine R., Jacobs B., Fähndrich M., Rustan K. R. M, Schulte W. and Venter H.,(2005) The Spec# programming system: Challenges and directions. VSTTE 2005.

Bhattacharya, P, Iliofotou, M, Neamtiu, I, Faloutsos, M. (2012). Graph-Based Analysis and Prediction for Software Evolution. Intl. Conf. on Software Engineering, pp. 419-429.

Chu S, Cesnik, B. (2001). Knowledge representation and retrieval using conceptual graphs and free text-document self-organization techniques. Int. Journal of Medical Informatics, 121–133.

De Moura L. and Bjørner N. (2008) Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2008, ETAPS Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337-340. Springer, 2008.

Gentner, D., & Forbus, K. (1991). MAC/FAC: A model of similarity-based retrieval. Proc. Cognitive Science Society.

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

Keane, M. T., Ledgeway, T., Duff, S. (1994). Constraints on analogical mapping: A comparison of three models. Cognitive Science 18, pp. 387-438.

Leino K. R. M. & Müller, P. (2009). Using the Spec# Language, Methodology, and Tools to Write Bug-Free Programs, Microsoft.

Leino K. R. M., Jacobs B., DeLine R., Chang B. E. & Barnett M. (2005) Boogie: A Modular Reusable Verifier for Object-Oriented Programs, FMCO 2005
Leino K. R. M. (2010). Dafny: An Automatic Program Verifier for Functional Correctness.
LPAR-16, April 2010.

McMillan, C. Grechanik M. and Poshyvanyk, D. "Detecting similar software applications," Intl. Conf. on Software Engineering, pp. 364-374, 2012.

Mishne, G., & De Rijke, M. (2004). Source Code Retrieval using Conceptual Similarity, Computer Assisted Information Retrieval (RIAO '04), pp. 539-554.

Monahan, R., O'Donoghue DP Case Based Specifications – reusing specifications, programs and proofs, Dagstuhl Reports Vol. 2(7) pp 20-21, AI meets Formal Software Development, 2012.

Montes-y-Gomez, M., Lopez, A., & Gelbukh, A. F. (2000). Information retrieval with conceptual graphs, Database and Expert Systems Applications, 312–321.

O'Donoghue D. and Crean, B. "RADAR: Finding Analogies using Attributes of Structure", Proc. AICS, Limerick, Ireland, 2002.

Ounis, I., & Pasca, M. (1998). Modeling, Indexing and Retrieving Images using Conceptual Graphs. Proc. 9th Intl. Conf. on Database and Expert Systems Applications, Springer.

Salton, G., Wong, A., & Yang, C. (1975). A Vector Space Model for Automatic Indexing. Communications of the ACM, 18, 613–620.

Sowa, J. F. (1984). Conceptual structures - Information processing in mind and machine.

Zaremski, A. M., Wing J. M. (1997). Specification Matching of Software Components, ACM Transactions on Software Engineering and Methodology, Vol. 6, No. 4, 333-369.