[C]**SHF: SMALL: SpecProg: Unifying Specifications and Code in Program Design**

1 Related Work

The cost of writing specifications is quite high and often the reason why the developers avoid writing specifications. However, if the specification is provided, it describes the behavior and intent of the concrete program. An alternative to this, specially specification written in natural language is using assertions [**?**], a concept borrowed from proof techniques. This can be considered as very simplified, but an effective effort towards unifying specification and coding. Interestingly, to the best of our knowledge, a very early work [**?**] has first questioned the separation of specification and other programming activities.

Another visionary, Donald Knuth [**?**] coined the idea of *literate programming (LP)*, and demonstrated the possibility of unifying a document formatting language and a programming language. Surprisingly, there were not many follow-up works for the improvement of this programming paradigm, possibly due to the lack of techniques to implement this idea in the past. The most well-known of them are [**?** **?** **?** **?** **?** **?** **?** **?**]. [**?**] provided an application of literate programming in object-oriented code, to develop computational electromagnetics (CEM) library. In this work, literate programming environment FWEB for C++ language helps the users by generating human-readable documentation in TEX format. [**?**] provides a chunk model that allows unlimited code and document types, and a theme model to provide multiple views of a given system based on different descriptions of a program aimed at different groups. [**?**] extends the literate programming tool using a document processor LYX to represent literate documents into tree representation. [**?**] describes the advantages of literate programming paradigm, and suggests that this paradigm should be used in the design phase of software development, as oppose to the practice of utilizing it in implementation and maintenance phases. VAMP [**?**] is a tool for supporting literate programming through auto generate modules. Compared to its prior work WEB, VAMP is not restricted to a single programming language. [**?**] proposed P-Coder, a tool for creating a better description of the program using literate notation. [**?**] provides a tool LP/ Lisp, which supports literate programming for a language with incremental development and testing like LISP. Unlike previous LP tools, which are suitable only for compiled languages that naturally have a step between writing and executing the code,

LP/Lisp supports the literate programming in running portions of the code. [?] states about the challenges of literate programming for new programmers, since it consists of 3 different languages: the implementation language, documentation language and literate programming glue language. This work proposes Ginger, a language that considers literate programming language as core feature and proposes G-expression transformation for representing code as well as documentation and linear connections. In conclusion, existing literate programming research focuses on extending the original paradigm to support object-oriented programming and visualization tools. Now-a-days, research for improving literate programming with large-scale repositories and current natural language to programming language inference techniques are needed. One of the most important component in literate programming, the translation engine from natural to programming language were predefined by a subset of natural language grammar and limited dictionaries of natural language words. So that, these prior works couldn't handle natural language descriptions which their grammar and words were not defined in the translation engine.

Another closely related direction is the area of structural programming through refinement techniques. Niklaus Wirth [?] first introduced the idea of gradually developing programs in a sequence of refinement steps to explore between preselection strategy and trial based stepwise construction. This notion has encouraged the programmers to concentrate more on designing *abstract* programs [? ? ?]. This can, in result, speed up the process of proving program correctness, since proving correctness of *abstract* program would suffice to prove the correctness of more detailed concrete program [? ?]. As modern software systems are inherently complex in nature and often remain in development stage in order to provide new and advanced features, incremental design has been embraced in both designing concrete programs [?] and specifications of the program [?]. Our approach is most similar to these works in a sense that we facilitate the specification-code duality that provides better abstraction and in addition expedites scalibility of verification.

The benefits of refinement paradigm has rallied researchers towards introducing different techniques to transform abstract to concrete programs, e.g., [?] proposes different alternative implementations to same abstract specification, [?] exploits domain knowledge towards program generation, [?] introduces the mechanism and benefits of recursive program synthesis from polymorphic refinement types. Additionally, various program synthesis techniques [? ? ? ? ? ? ? ? ? ? ? ?] have been observed to answer open ended questions on combining logical specifications, natural languages

to encourage an unifying programming interface [?]. Generally, these works proposed solution that suited for specific type of programming and specific programming language. One of most well-known application in this area is WolframAlpha [?]. This work provided a computation engine that allows users to have the calculation output from well-known algorithms by their description in queries that commonly contain the name of algorithm and input arguments. [?] proposed a technique for generating assembly code from users' natural language description for industrial robot actions. This work applied state-of-art NLP technique for semantic parsing and syntactic parsing to get the set of predicate-argument, which represent a sequence of tasks from the description, then mapping each predicate-argument to related action-object of the simulated word. The action-object is predefined and stored in a unified architecture. [?] designs a new language Quasi-Natural language that focuses on knowledge representation, which contains data structures and suitable operations. Thís language provides support for natural language which are expressed in certain lexical and syntactic rules. [?] provides a program synthesis approach for domain specific language (DSL) by inferring a dictionary relation over pairs of English words and DSL terminal semi-automatically. [?] proposed another approach for program synthesis, which considers synonym problem and provides modules to handle each type of statements in a programming language. [?] focuses on the control structure of the programming language and provide an approach for using type dependency in natural language sentence to infer the its control structure. [?] proposes an heuristic approach to identify basic elements of object-oriented programming, including class name and variables/attributes. This work uses new heuristic rules manual from a training data set, then apply and evaluate the correctness of these rules in another set of natural language description corpus. SWIM [?] uses data-driven approach to synthesize code snippet from users queries in two steps: extracting ranked APIs set for each query and, then, select structured call sequence for each API. The query to API model was learnt from *clickthrough* data of Bing, while structured call sequence was learnt from Github code corpus. Pegasus [?], which is a first step towards implementating the ideas of naturalistic types [?], designs their approach in following pipeline: reading natural language, generating programming code and expressing natural language. One of most recent work [?] combines ideas of parsing technique from natural language and type-directed synthesis, program repair from programming language to generate SQL query from description. These highly advanced works can be leveraged to synthesize concrete programs and motivate programmers in regard to the practice of writing specifications.

Additionally, Dunsmore [?] analyzes the programming efforts and state that ease of modification is related to the data communication and live variables. Therefore, replacing code by specification whenever possible can ease in terms of the programming effort required in software maintenance and evolution. Leino [?] supports varying level of abstraction. The authors theorize that higher level abstraction can focus on the intent of the design, whereas the lower level abstraction can centralize on optimization and other detailed factors. SpecProg can provide similar advantages by supporting specification and code in unison.

As Knuth [?] advocated that one of the primary goal behind bringing forth the specification is understanding the software itself. It enables multifaceted advantages, e.g., facilitate maintaining legacy code, reuse of code etc. Naturally, the specification written in natural language is more human-readable compared to formal specifications. However, specification in natural language form does not suffice for verification purposes. If developers take a manual approach to transform between natural and programming languages, it will cost both in terms of effort and time. Besides, this can be often error prone, since the inference requires developers to have domain knowledge in both types of languages. To overcome these challenges, approaches to automatically derive natural language from programming language and vice versa have been designed.

To support better understanding of the source code by documentation, some approaches aimed to automatically generate some restricted type of natural language, e.g., generating pseudo code [?] and code comment [?]. [?] applied statistical machine translation (SMT) approach to generate pseudo-code from Python source code. This work proposed an improvement for original Phrase-To-Phrase translation commonly used in Natural Language Processing (NLP), by providing a Tree-To-String translation approach. They built a parallel corpus at method level, where, the source language reflects Abstract-Syntax-Tree, and target language reflects pseudo code of the same method. This work limited the generation process to statement-by-statement, meaning that the pseudo-code was generated for each statement in the source code. This process is insufficient, however, for generating pseudo-code in large software project environments that contains more than thousand lines of code. [?] proposed an approach to extract the comments from Java methods that summarizes Java methods' content. This work also focus at method level, by the extracting a set of important central lines of code (s-unit). Then it generates the summary for lines in s-unit using a *software word usage model* constructed for each method.

The inference from natural language to programming language seems to

be tackled more compared to the other direction of reference. There are researches on analyzing Natural Language grammar versus Programming Language grammars and researches on approaches for the inference of Programming Language from Natural Language.

To analyze natural language and programming panguage grammars, one of the starting point was made in 1977 by L. Miller [?]. This work proposed *Natural Language Programming*, which analyzes natural language programs to have semantic representation of 1000 unique-words. [?] analyzed the differences between programming language grammars, which were built by Context-Free-Grammar versus natural language grammar. The experiment shows that using natural language grammar can be used as an alternative for humans to learn the fundamental of programming, which is more attractive and promising. [?] analyses characteristics of natural language by the mapping programming language. This work proposed the idea of naturalistic types, which contains 4 elements: concept, properties, quantities and conditions. In game programming development, [?] argued that entire natural language will not be a viable option for coding. Based on the experiments, the authors suggested some design rules for making use of natural language, among these are: constrain the programming language to minimize syntax error and the natural language used should be consistent with typical everyday use and avoid unnatural syntax or phrasing. Contary to these approaches [?] analyzed source codes and provided experiments to identify which patterns in source code can lead to misunderstanding in software development.

There have been some other ongoing research [? ? ? ? ?] on motivating towards *naturalistic* programming languages. Efforts [? ?] have been made to address the one of the main issues in naturalistic programming, which is dealing with ambiguity. Clark debates the trade-off between *naturalness* and *predictability* and proposes a synthesis of embedded deterministic core in naturalast controlled languages. However these efforts only tries to bridge the gap between natural and programming languages, contrary to our objective of unifying the specification and code as a single programming paradigm. Swartout advocates intertwining of specification and implementation as oppose to completing the specification process in advance. A closely related work is [?], that takes a program template together with rich functional specification and produces concrete program with proof artifacts. While these works can be leveraged towards specification-code transformation and refinement, these are only partial focus in the path of unification of specification-code.