A Theoretical Concept: Towards Mathematical Declarations of Code Intentions

Athanasios Tsitsipas, Lutz Schubert
Institute of Information Resource Management
University of Ulm
Ulm, Germany
e-mail: {athanasios.tsitsipas, lutz.schubert}@uni-ulm.de

***Abstract*—"The whole is more than the sum of its parts" (Aristotle). Current imperative languages do not allow a program to be simply broken up (decomposition) or to merge several parts of a program, but demand appropriate knowledge and manual effort. The idea behind is to transfer methods from mathematical combinatorics to standard programming models to enable the distribution of a task across multiple heterogeneous resources. This approach allows distributed, heterogeneous resources to be treated as an integrated platform, with no hassle of adaptation for the developer. In this paper we propose and discuss a theoretical framework, with which the correctness of the code can be guaranteed with automated (de-)composition and adaptation. This will lay the groundwork for new programming methods that will allow code to be more fully understood, analysed and modified. This is relevant for all areas that develop and use software.**

Keywords–Software Engineering; (De-)composition; Group Theory.

I. INTRODUCTION

The world functions like a well-tweaked clock; it constantly moves and changes as we struggle to keep pace with it. We represent our world mathematically in order to explain, predict and reason with it - in other words, to scientifically deal with it. We base on theorems, axioms and lemmas that will eventually enable us to break down a problem into simpler steps, commonly understandable. Everything flows (Heraclitus) in the world of Information Technology(IT) with new types of resources and applications arriving on a daily basis, making it hard to keep pace. Software Engineering is still based on the principles of Alan Turing, and we are still developing hardware-specific programs. However, as more manufacturers specialize in Integrated Circuits for Dedicated Devices, variations in platforms have increased. Thus, different compilers become necessary, *e.g.,* convert C / C ++ into code optimized for the respective platform. This process is time consuming and requires that the code itself should be adapted to the target platform. Our goal is to create a theoretical framework that establishes concepts and methodologies, harnessing the power of mathematics as means to control, analyse and reason over the dynamic elements of a modern IT environment. We believe that we can define a program once and execute it on and across multiple environments, with no exertion of adaptability for specific resource types and environments. We should be able to add, subtract or alternate functions and/or features in an IT environment on demand, without having to worry explicitly about correctness and feasibility.

Within this short paper we will first examine the background information in related areas of software engineering (section II). In section III we outline the approach, where we propose a theoretical framework that describes the dependencies that arise in the decomposition and merging of subtasks from a mathematical point of view and thereby ensures the correctness of the resulting tasks. We conclude (section IV) with a short summary and future work.

II. BACKGROUND

In current software development, the following things cannot be done properly with standard programming models: correctness of code (de)composition, multipurpose deployment, and easy execution. Currently, code (de)composition can only be achieved using well-defined patterns, which implicitly constrain execution to pre-defined use cases, logic and situations (infrastructures). This is because a compiler cannot understand a program automatically (so-called Halting problem). Nevertheless, we need to be able to automatically change the algorithmic logic when combining or separating functionalities, as well as, when using new resources. The functional behaviour needs to remain the same even though we change the algorithm (the logic). By using predefined code patterns, such as executing loops sequentially, or using map-reduce. This can partially be achieved by the compiler, if sufficient information is given (*i.e.,* if it fits a certain pattern), but not generally in a well-defined fashion. By applying group theory to combine logical elements, we could in principle develop a generic method for algorithmic (de)composition and adaptation, thus not solving, but certainly reducing the Halting problem considerably.

We change the way of standard programming with a mathematical description that abstracts from the algorithmic intention of the code, and not use programming models to only parallelise it (*i.e.,* Skeleton, TBB, Cilk). In order to utilize arbitrary resources in changing environments, we have to be able to break up (decompose) a function into a combination of functions, aligned to the available resources. Something similar has been attempted with the General Problem Solver [1] and is proven to be NP-hard. The General Problem Solver tried to find an optimal path over an infinite graph, whereas the group theory defines the combinatorial behaviour that in itself can generate a graph (and hence path) through a complex function (much like solving an equation can be represented by graphs).

Non-functional properties of execution (such as performance) depend on the available resources, which may change unpredictably. However, if we can express the non-functional properties as a projection of the function, then any decomposition of the functional declaration will be applicable to its projected functions too, and therefore to the non-functional properties. Since execution characteristics (non-functional properties) will change with the resources used, we need to exploit the behavioural patterns for code execution in alignment with the resource characteristics to find the best match between intended properties and the way this pattern executes on said

 ISBN: 978-1-61208-655-2

resource. In our proposal, we exploit the (de)composition capabilities of mathematics, *i.e.,* that the same function can be expressed as different compositions of functions - this decouples the algorithmic (de)composition complexity from the actual functional intention and can be solved through a set of reference transformation rules, as demonstrated in POLCA [2]–[6].

III. Principal Approach

Concepts from the realm of mathematics in abstract algebra, and more specifically from group theory form the basis for the proposed work. The theory of groups occupies a central position in mathematics to delineate and control the space occupied by any polynomial function. The definition of the group is a well-formulated method in mathematics and can be applied in many domains, including arithmetic, geometry, but even beyond in biology, chemistry, physics [7]. A group is a set of elements, equipped with an operation $\star$ that comply with certain algebraic laws (associativity). If we combine two elements in a group (add two integers), the result is also in the group. Mathematical groups are not constrained to simple elements (such as numbers), but can be applied to complex objects. In the context of this work, we will try to transfer these concepts to programming languages, making computable functions complex objects that can serve as elements in a group. Note that this goes beyond the standard algorithmic definition of a function. By trying to perceive the infrastructure and the applications as a mathematical equation, we consider them in sets and combinations of elements.

We impose approaches that were developed as early as the 1960s, but were not pursued because of their difficulties in implementation: the mathematical declaration of the problem instead of the imperative-algorithmic one. What is special about the mathematical versus the algorithmic declaration is that the same problem can be solved in different ways, namely (1) by **mathematical transformation**

$$a^2 + 2ab + b^2 = (a+b)^2 = (b+a)^2 = (a+b)(b+a) \quad (1)$$

and, (2) by **converting the formula into various algorithmic forms**:

```
double binom(double a, double b)
   return (a*a+2*a*b+b*b);
or return (exp(a,2)+exp(b,2)+2*a*b);
```

Thus, the mathematical declaration is a superordinate definition of the overall behavior of the application, which can be broken down and distributed into different subtasks. Moreover, by using the transformations of the group, the correctness of the solution is always guaranteed, *i.e.,* in the given case by the commutativity, associativity and distributivity of + and * over $\mathbb{R}$ in the above examples. The relevant point is that this applies to every mathematical group, regardless of the definition of the operation and the space (as long as they satisfy the basic conditions) [3], [8].

The composition and in particular decomposition of code in Software Engineering is an NP-hard task. The challenge here is how to apply concepts from group theory to software engineering to enable the distribution of functions across multiple heterogeneous resources. Such an approach has never really been attempted before and though the principle may seem obvious, there are no well-defined methods yet and the consequences of any such approach must still be explored: Due to the inherent complexity of the problem there is an increased risk that the methodology may be applicable only under limited conditions. However, it will open discussions and will be an intriguing topic and of relevance for scientific research and industrial purposes.

IV. Conclusion and Future Work

With the proposed idea, it will be possible to deal better with complex applications that utilize multiple resources, which is currently possible only with a tremendous amount of effort. While programming and adapting programs is still a hard task, we envision a programming model where a functional abstraction through intentions of code will be realised, thus enabling us to overcome the problems arising from (de)composition of algorithmic logic (Halting Problem). As future steps, by developing new compilation methods and a novel code declaration to analyse, rearrange and manage software and finally ensure correctness, we will initiate new discussions for a path to rethink the way of programming. One major challenge faced by the idea consists in the potential combinatorial explosion: by applying group theory, a code can be arbitrarily combined and segmented leading to a potential infinite number of solutions (or rather: equivalent functions). To counter this, new methods to constrain and guide the search space will have to be examined, relating to current compiler techniques.

References

[1] A. Newell, J. C. Shaw, and H. A. Simon, "Report on a general problem solving program," in *IFIP congress*, vol. 256, 1959, p. 64.

[2] The mathematics behind polca. http://polca-project.eu/downloads/presentations/33-math-behind-polca/file. Accessed: 2018-04-01.

[3] J. Kuper, L. Schubert, K. Kempf, C. Glass, D. R. Bonilla, and M. Carro, "Program transformations in the polca project," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016, pp. 882–887.

[4] S. Tamarit, J. Mariño-Carballo, G. Vigueras, and M. Carro, "Towards a semantics-aware transformation toolchain for heterogeneous systems," in *Program Transformation for Programmability in Heterogeneous Architecture Workshop (PROHA)*, 2016.

[5] D. R. Bonilla, C. W. Glass, and J. Kuper, "Optimized polynomial evaluation with semantic annotations," in *Program Transformation for Programmability in Heterogeneous Architecture Workshop (PROHA)*, 2016.

[6] S. Tamarit, G. Vigueras, M. Carro, and J. Marino, "A haskell implementation of a rule-based program transformation for c programs," in *International Symposium on Practical Aspects of Declarative Languages*. Springer, 2015, pp. 105–114.

[7] E. P. Wigner, "The unreasonable effectiveness of mathematics in the natural sciences," in *Mathematics and Science*. World Scientific, 1990, pp. 291–306.

[8] L. Schubert, J. Kuper, and J. Gracia, "Polca–a programming model for large scale, strongly heterogeneous infrastructures," *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, vol. 25, p. 43, 2014.

 ISBN: 978-1-61208-655-2