Imperative Functional Programming

Software Engineering with I4

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

Keith Jeffery
Keith G. Jeffery Consultants
Shrivenham, UK
Email: keith.jeffery@keithgjefferyconsultants.co.uk

***Abstract*—Applications need to be constantly re-developed for new devices and infrastructures, and to address new user needs. This leads to an increasing maintenance cost that only large-scale companies can afford. The problem with traditional Turing based programming models is that algorithms cannot be easily adjusted and thus bind the application to an environment. In this paper, we discuss how mathematical definitions can be used to not only describe algorithms, but specifically to allow their transformation and (re-)generation to principally address different infrastructures and requirements at considerably reduced effort.**

Keywords-software engineering; I4; abstraction; declarative programming; imperative programming.

I. INTRODUCTION

Modern infrastructures are defined by a degree of heterogeneity and complexity never encountered before. Myriads of new devices are connected to the internet and want to be used and controlled. Each infrastructure and resource have their own specific characteristics that are difficult to fully exploit without adjusting the application to it. Modern resources may not even be Industry Standard Architecture (ISA) compliant. Hence, such new devices demand significant changes in existing software and a large part of the software industry is already just occupied with ensuring that code runs on and with these new devices.

Traditional programming models based on Turing's concepts [1] are close to the hardware organization. In order to achieve best performance and meet the desired constraints best, every new ISA and hardware organization therefore necessitates a re-thinking and hence re-development of the algorithmic structure to meet the hardware specific characteristics. This leads to significant cost for code maintenance and portability, leading to more than 75% of the development cost [2][3]. Implicitly, smaller companies with new software ideas will not be able to stand the growing pressure to fix bugs, adapt the software to new devices, etc., whereas the pressure on big companies from small innovative, but un-sustainable ideas, grows constantly.

To overcome these constraints, software engineers have always been working on ways of abstracting from the hardware and thereby trying to get closer to the natural way of specifying tasks. However, in general all new models "just" build up on the existing constraints, thus incorporating and wrapping them, rather than addressing the problem directly. In the following we will investigate how developers think about software and how they go about addressing specific objectives. Based on these observations, we will try to derive more flexible software engineering principles that will allow for higher portability and adaptability to different platforms. We will demonstrate that by exploiting intrinsic mathematical properties of code and its properties, we can emulate the developer's behavior in code transformation and adaptation, whilst maintaining or addressing specific properties. The work presented here builds up on discussions in the European Commission's Cloud Computing Expert Group and documented in [4][5] which include any background and related work with respect to the approach.

This paper is structured as follows: in Section II, we will examine the typical software engineering principles and try to derive a generalized model from this. Our principles are based on the assumption of mathematical equivalence to code, which we will examine in Section III. Section IV will try to apply this assumption to the full software engineering principles. We discuss the approach in the concluding Section V.

II. THE SOFTWARE DEVELOPMENT PROCESS

Developers are guided by four main principles in their programming process, which we call the four "I"s [5][6]: (1) the *Intention* behind the application, i.e., what the developer actually wants to achieve with it; (2) the *Information* used, processed and generated by the application; (3) the *Incentive* defines the mode in which the functionalities are to be offered, i.e., fast, reliable, etc.; and finally (4) the *Infrastructure* on which the application is to be executed. Let us see how a developer makes use of these four parameters when programming a (new) software:

A. Intention

All software starts with an intention, i.e., with an idea of what the application is supposed to do, once finished. Most programmers already think in terms of steps and procedures at this point, but this is just because of their experience, as can be easily observed on programming beginners. In itself, the intention is not bound to any algorithm or process other than by logical constraints: to make a banana milkshake, it is sensible to switch on the blender after banana and milk have

 ISBN: 978-1-61208-732-0

been added, but the order of banana and milk are independent, as is the amount, the type, flavours, etc..

In principle, Turing has already shown that any solvable problem can be solved in a near infinite number of ways, if the individual steps are small enough (think of *how* to add the banana). The process is not prescribed at this point, though the principle steps involved will be known to most humans, though everyone will execute it differently. The things relevant to know in this context, are only the *principle steps involved,* the *logical constraints* and *relationships.*

B. Information

Data is one specific form of representing information, and as any communication scientist will know, information is frequently lost by converting it into data. Vice versa, extracting information from data is not always possible and frequently requires human intervention (think of a book as data and the information you extract by reading from it).

For a software engineer, finding the right way of representing information is a challenge on multiple aspects, as it will (1) define the data structure, thereby (2) influencing the algorithmic behavior and (3) constrain the processing and reusability. In general, data is hardly ever the desired outcome of an application, but the information behind it. Even large scale, data-bound applications, such as fluid simulations actually just want to identify where and what kind of turbulences occur, not the pressure and velocity at any given point – we are just constrained in the way that we compute said information without breaking it down into (particle) data. This relationship is complex and requires considerable expertise by the developer.

C. Incentive

Incentive may seem the least intuitive at first, as it is something that most developers and users specify only indirectly. By nature, the incentive is closely related to the intention, yet changes the "flavor" of the latter ever so slightly, for example if the application is supposed to be fast versus reliable. Incentives can create the most contradiction and confusion, so that developers will have to find the best middle way between all requirements posed towards them.

The incentive is the main deciding factor for generation of the algorithm, as the developer will have to choose whether to generate a parallel code, a service-oriented or modular approach, whether an algorithm is reliable, fast, storage-consuming etc. Traditionally, software developers are trained in a specific direction and will make the choices intuitively, such as is the case for HPC programmers. Thus, to interpret and "enact" an incentive, we need to know the *properties of an algorithm.* This is a highly theoretical field and, as we shall see, poses many obstacles for automated code generation.

D. Infrastructure

Obviously, the final algorithmic choices are made when the target infrastructure is known. Obviously, the incentive already plays a large role in selecting the target infrastructure and vice versa – for example a complex code that needs to be executed as fast as possible will probably have to be parallelised and will have to run hence on a parallel infrastructure; whereas an application with multiple users will probably be destined for a cloud-like web infrastructure, etc.

In this final step, the final code details will be decided, leading to the final algorithm that can be compiled. To realise this, the developer has to know something about the *relationship between hardware properties* and *code behaviour.*

E. Summary

The four parameters (Intention, Information, Incentive, Infrastructure), are sufficient to describe all aspects that guide a developer from idea to code (see Figure 1).

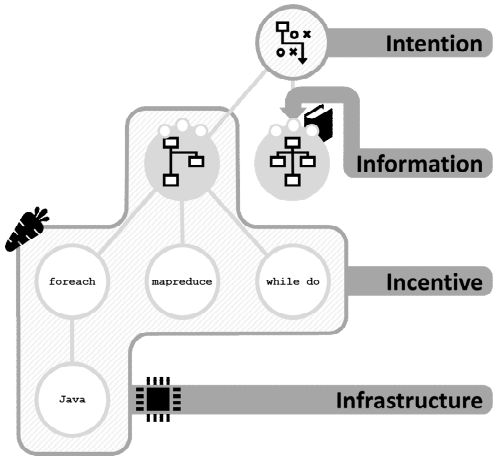

Figure 1. The Software Engineering process.

How can such parameters and the necessary background knowledge for transformation be encoded? We built up on Turing's main principle, namely that computer programs are *mathematically solvable problems* and hence are mathematically expressible. This means for us that they are hence also treatable as mathematical objects, including all according transformation rules. Based on this assumption, most computable problems and therefore applications should be transformable the same way as mathematical formulas. As an implication, if we can express the (human) software engineering process mathematically, we can also use according rules to perform the transformation steps.

III. PRINCIPLES OF I4

The idea here is based on the following main principles: (1) any mathematically expressible problem can be converted into an algorithmic structure; (2) mathematical expressions can be treated mathematically; (3) algorithmic structures can be distinguished by their structural properties, which in turn relate to the specific properties of the code.

As a simple example, let us assume we have a simple task (*Intention*) to count the number of elements in a set of objects. Mathematically, we can define count recursively:

$$count(P) \equiv |P| := \begin{cases} 1 \; if \; \forall p \in P : P \backslash \mathrm{p} = \emptyset \\ |Q| + |R| ; Q \subset P ; R = P \backslash \mathrm{Q} \; else \end{cases}$$

We can resolve this function by counting the recursively generated leaves (see Figure 2).

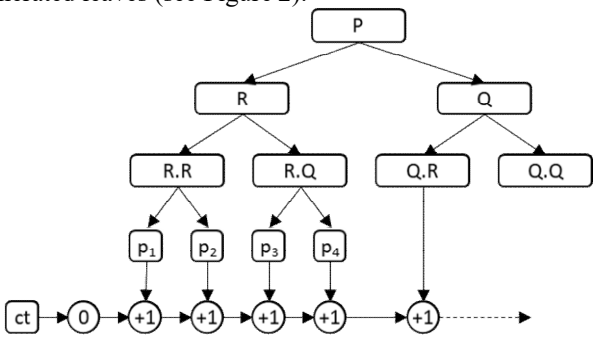

Figure 2. Recursive solution of "count(P)".

It is obvious from Figure 2 that the code can be distributed, serialized, etc. – in other words, we can associate different properties with the structures. What is more, the pattern can be easily described in a higher-order-function as:

```
count(P) ≡ foldl (+1) 0 P
```

which can be realized as a for loop over all elements in P – no matter how P is organized. With this definition, we can also apply simple transformations which in turn affect the code behavior again. For example, we know that

```
count(P) = count(Q)+count(R) for R∪Q=P
```

and thus implicitly

```
count(P) ≡ foldl (+1) 0 P ≡ foldl (foldl (+1)) 0
(R Q)
```

which represents two consecutive loops. This leads to an execution cost of $p = r+q$ operations (|P|=|R|+|Q|) and thus the same as without transformation. However, as evidenced by Figure 2, we can easily apply a further transformation

```
foldl (+1) 0 P ≡ foldl (+) 0 (map (foldl (+1) 0)
(R Q))
```

which is fully equivalent according to the base properties of *foldl* and *map*, but obviously can now be executed in parallel (see Figure 3) and thus leads to operational cost of *max(r,q)+1* which is considerably lower than *r+q*.

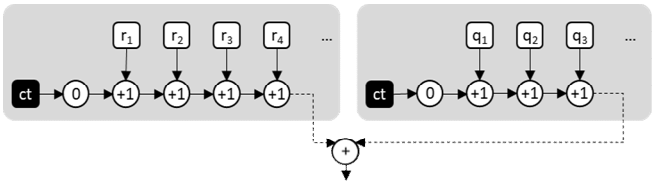

Figure 3. Recursive solution of "count(P)".

Though this is clearly a very simple example, it still shows how a descriptive task ("count") can be (1) converted into an algorithmic structure which (2) can be transformed on a mathematical basis so as to (3) change its properties, such as computational complexity and degree of parallelism.

IV. Application to Software Engineering

With this base principle in place, we can examine how the full software engineering process, as described in Section II, could look like based on mathematical principles. Building up on the "count" example, we can investigate how to count unique elements following the software engineering cycles above, for example to perform statistical evaluations:

A. *Specifying the Intention*

As a developer, we have immediately multiple ideas and algorithms in mind how to count unique elements in a given set (array, list) and thus this can serve as a full specification for an application. We can thus define:

Intention: count unique elements

To convert this into the form of mathematical specifications that can be reasoned over, we need to first of all specify the relationship between the "intentions", as "unique before count" with a place holder for the set, i.e.

```
count ∘ unique (P)
```

where *count* is defined as above and

$$unique(P) := \{\forall p_i \in P: \nexists p_j \in P, i \neq j: p_i = p_j\}$$

It is important to stress here that even though an equality operator (=) is used in the definition, this operation may differ completely between types of objects (i.e. Information, see below), just as we could override operators in C/C++. As long as we uphold all equality properties, this definition holds true, even if only partial aspects are used. This is important for the developer, as the code will change substantially with definition of the data structure.

Notably, again we can split P into subsets R and Q, so that P=R∪Q with R=P\Q, leading to

$$unique(Q \cup R) :=$$
$$\{\forall p_i \in (Q' \cup R'): \nexists p_j \in (Q' \cup R'), i \neq j: p_i = p_j\}$$
$$\text{with } Q' = \{\forall q_i \in Q: \nexists p_j \in Q, i \neq j: q_i = p_j\}$$
$$\text{and } R' = \{\forall r_i \in R: \nexists p_j \in R, i \neq j: r_i = p_j\}$$

This looks very similar to a direct split, yet it will be noticed that by default Q' and R' will be smaller than Q and R, respectively, thus reducing the workload for the final uniqueness test. Since we also know that *unique* must be executed before *count*, we can specify a general task-flow on basis of the knowledge so far, such as depicted in Figure 4.

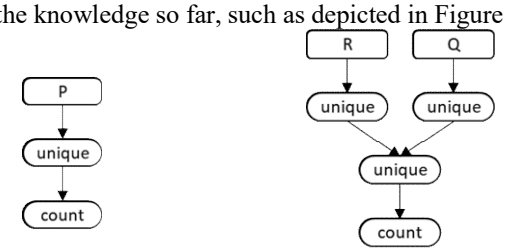

Figure 4. Simple task flow for "count unique members of P" (left), respectively the options for splitting P into R and Q (right).

Any reader with development skills will immediately get an idea for the code just from the specifications above – in particular given the capabilities of most higher order languages for operator overloading. From the specification, we can see that all elements need to be compared against each other (see below for optimization), and that the elements of the resulting set then needs to be counted. We can also already see that both operations can be combined

 ISBN: 978-1-61208-732-0

and executed in different distributions, depending on context (*Incentive*). At this point, a pseudo-code could look like this:

```
Q=P
foreach (q in Q)
  foreach (p in (P\q))
    if (q==p) Q=Q\q
ct=0
foreach (q in Q)
  ct++
```

Note that the code would not execute for multiple reasons, among others because we manipulate the set during traversal. More correctly we would temporarily save the values and remove them in the end – the behavior is nonetheless sufficiently defined at this time.

B. *Influence of Information*

It has already been noted that information will greatly influence the code definition above – this is already obvious by the simple circumstance that "uniqueness" is a highly subjective and philosophical notion. We can for example specify that two people are identical if they have the same tax id, or that two objects are the same if they have the same shape and color, etc. As a developer, we would specify the object as a complex struct and overload the equality operation to allow for such behavior. However, as a High Performance Computing (HPC) or Embedded Systems developer, you would probably point out that this structure is not aligned to data access, consider:

```
struct molecule {double px, py, pz, w }
foreach (mol in molecules)
  mol.w = mol.w*c
```

As can be seen, this leads to a stride in memory usage and thus to an 75% underutilized memory, which in turn affects cache performance, leading to 4 times more cache misses than necessary (see Figure 5).

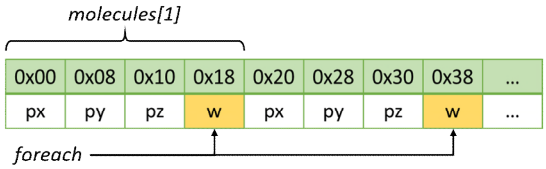

Figure 5. Memory organisation for an array of structs.

By converting this array of structs into a struct of arrays, we can easily improve memory utilization and thus cache performance (see Figure 6):

```
struct molecules {double px[], py[], pz[], w[] }
foreach (weight in molecules.w)
  weight = weight*c
```

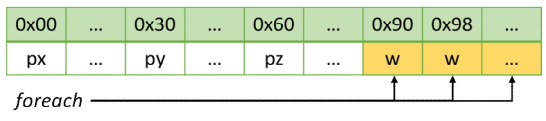

Figure 6. Memory organisation for a struct of arrays.

With the decision for a specific layout, the developer has constrained adaptability of the algorithm considerably at this point. Few compilers support the conversion from array of structs to structs of arrays and vice versa and will always need additional information by the developer to do so. By exploiting *Information*, we do not specify the layout yet – it is in fact often considered a weakness of functional programming that memory layouting cannot be influenced by the programmer [7]:

$P \subset People$

People have *name, location, …*

By adding a specification that we consider two elements in people as identical if they have said the same names:

$\forall p1, p2 \in P: p1 = p2 \Leftrightarrow$

stringmatch(name of *p1, name* of *p2)*

We thus have a data structure without a concrete layout and we can easily see that both memory arrangements (see Figure 5 and Figure 6) are possible with this definition.

C. *Setting the Incentives*

Notably, the memory layout is directly related to the incentive behind it: a struct of arrays may be more sensible in situations with high performance requirements, whereas an array of struct is sensible if the work is distributed, i.e., when different operations may be performed on the array. Thus, with defining the incentive, we make a concrete instantiation choice for parts of the algorithm, which is closely related to the infrastructure impact, below:

The *incentive performance* can be seen as a projection function from the data access structure and the executional pattern to cost. We have already indicated above that the operational load can be roughly derived from the number of operations resulting from the size of the set. In algorithm theory, we generally assess the order of complexity based on the execution patterns [8] which gives us an indicator for workload and thus performance of an algorithm. Communication overhead can be assessed through data size, messaging frequency and network properties (see infrastructure). Notably, this provides only relative information, as the size of the data set will still affect distribution, degree of parallelism, etc., but it already allows to distinguish between different choices.

We can see that the parallel implementation of the count ○ unique function leads to a significant reduction of operational load (per processor). Since we also just access the *name* property, we can not only reduce the memory load through a struct of arrays, we can even completely discard all other properties associated with *People* (though this obviously depends on the intention in the first instance).

To realise this, we need to associate the data access cost to a complexity function similar to the algorithmic operation load. Obviously, this is directly related to communication modalities and can thus be assessed similarly, where the cost must be related to non-accessed areas. In other words, we can use indicators, such as ratio between accessed and non-accessed data size based equally on the access patterns (see Figure 3), as well as on the data structure decisions.

D. *Specifying the target Infrastructure*

Only when the task flow is mapped onto a system model can the Incentives be fully assessed and properly matched.

 ISBN: 978-1-61208-732-0

Traditionally, infrastructures are modelled as network graphs, where each node represents a resource and each edge a connection between resources. This allows distribution of deployment graph and thus analysis of the impact on performance, respectively on other Incentives. Given the scale and complexity of modern systems, this approach is not feasible for real world problems. Furthermore, it is as yet unclear, which resource characteristics impact on application properties how –simple properties, such as number of cores, are used, or the hardware is profiled for an application and said profile is then used instead of characteristics.

The I4 model combines these two aspects and relaxes the characteristics definition. We foresee that future machine learning methods building up on the profiling principles devised, e.g. in CACTOS [9], that will automatically categorize profiling information according to the resources used and thus generate more meaningful properties. For now, we assume a relationship graph similar to a network model with annotations meaningful in relation to the Incentives and Intentions, here such as: *multicore* or simply *number of cores,* and *bandwidth, latency,* etc. We can thus define, e.g.

```
User.Dev = {Smartphone}
DB1.Dev = {Virtual, MySQL, 4 cores, ...)}
DB2.Dev = {Virtual, MySQL, 4 cores, ...)}
G = ({User.Dev, Internet}, {(User.Dev, Internet)})
Gt = ({DB1.Dev, DB2.Dev}, {(DB1.Dev, DB2.Dev)})
Gp1 = ({DB1.Dev, Internet}, {(DB1.Dev, Internet)})
Gp2 = ({DB2.Dev, Internet}, {(DB2.Dev, Internet)})
```

This information allows us to generate a simple network graph such as depicted in Figure 7. Comparing this to the potential instantiations of our task graph (see Figure 4), we can immediately recognize the potential task distribution, respectively how the work could be split between resources. This is principally a "simple" graph matching task, bearing in mind that multiple solutions are valid, so greedy matching approaches will be sufficient [10].

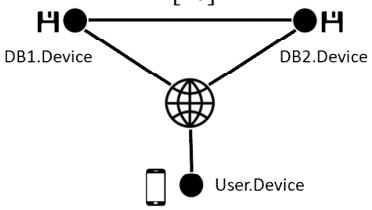

Figure 7. Target infrastructure network graph.

Based on the *performance* incentive, we would try to parallelise and reduce the communication between points, which gives us a general guide to the matching strategy.

E. Generating the Algorithm

We now have all the relevant information in place that would allow a developer to generate an algorithm that meets the specified requirements (the four "I" s). An experienced developer will also see that some of the choices made above will lead intuitively to sub-optimal solutions. Specifically, the separation of unique and count seems less than optimal, since the loops could be fused. Now, we should note at this point that the approach suggested here does aim at replacing existing compiler techniques and, e.g. loop fusion can also be performed by most compilers. Nonetheless, we will show in the following how such techniques can be respected and will influence the transformation choices and outcome.

Following the strict usage of all information, we can derive that if both database sources should be considered, the best approach treats the databases DB1 and DB2 as R and Q, respectively (see Figure 4 (right)). We can also see how mapping to the infrastructure allows different distribution of *count* to exploit task parallelism and communication delays (see Figure 8).

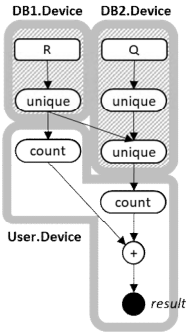

Figure 8. Task flow (see Figure 4) mapped to target infrastructure (see Figure 7).

This analysis is straight-forward and can be directly derived from the individual task-flow graphs that can be spanned by such simple transformations as (1).

We have already seen in Section IV.A, how algorithms can be generally derived from Higher Order Functions. In general, this relationship is more or less straight-forward, though we must bear in mind that different algorithmic presentations exist. For example,

```
foldl f a P
```

can be expressed as

```
z=a; for (i=0; i<P.length(); i++) z = f(z, P[i]);
```

or, since no order is given by foldl, also as

```
z=a; foreach (p in P) z = f(z, p);
```

Obviously, we could use while loops, serialise the execution, recurse it, etc. Similarly,

```
map f P
```

can be represented as

```
for (i=0; i<P.length(); i++) z = f( P[i]);
```

and so on. So, with the definitions for *count* and *unique* as discussed, we can derive algorithms for the individual target resources, e.g.

DB2.Device:

```
Q'=Q
foreach (q1 in Q)
  foreach (q2 in (Q\q1))
    if (q1==q2) Q'=Q\q1
Q''=Q'
foreach (r in R')
  foreach (q3 in Q')
    if (r==q3) Q''=Q'\q3
```

It will be noticed that due to the symmetry of equality, not all elements need to be checked with all others, but in fact that if *q1=q2* then *q2=q1* and hence the algorithm for *unique* can be changed to

```
if (q1==q2)
 Q'=Q\q1
 Q=Q\q2
```

It is important to note that *q2* is removed from the search set, due to equality and not from the result set. If we follow the whole process through for each resource, task and all relationships, we thus can generate a task-based execution pattern such as depicted in Figure 9.

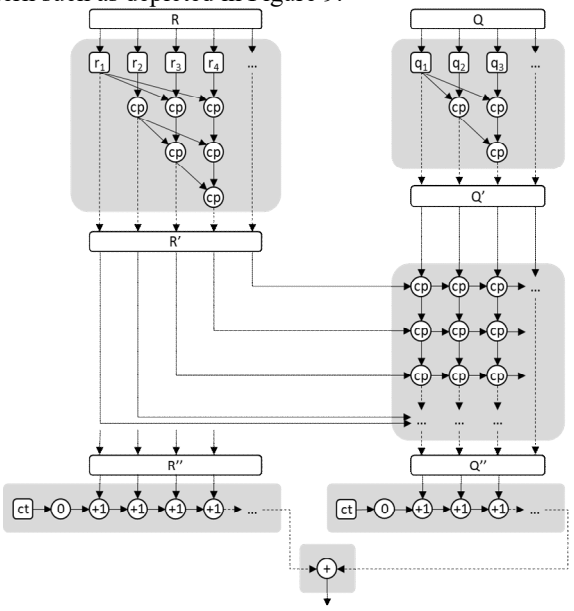

Figure 9. Full flow graph for *count* ∘ *unique* (*R*∪*Q*).

By investigating the dependencies between operations, we will also notice that, in principle, counting can be directly merged with testing for uniqueness (see Figure 10). Even though beneficial for cache access, the actual operational load does not change this way though and it is up to the service owners, which versions they prefer – in principle, the transformation processes described here can derive principally any viable distribution, leaving it up to the developer to make a final decision (or just choosing one).

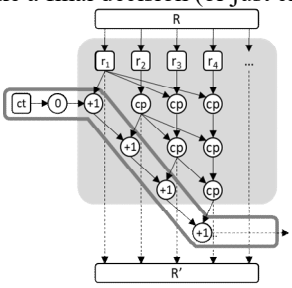

Figure 10. count and unique combined.

V. CONCLUSIONS

In this paper, we have presented an approach to generate *Incentive*- and *Infrastructure*- adapted code using a specification of the *Intention* of the application and the *Information* to be processed. The principles build up from the initial discussions under "Complete Computing" [5] and are still work in progress. For example, while the general principles are clear, the full assumptions and scope of applicability still need to be fully developed and analysed. Even though the same methods will apply for more complex application specifications, the computational complexity rises considerably, necessitating the introduction of metrics to guide the transformation process. Such metrics can be derived from executional properties and backpropagation over the decision tree – this is currently under investigation. Additionally, due to the status of the approach industrial applications cannot be addressed.

Another question obviously arises from problems that are not directly mathematically expressible. Many algorithms have been developed that are basically a set of tasks to be performed, much rather than solving a mathematical problem as such. It can be argued that any computational problem can still be expressed mathematically, though the question would be whether the additional effort is worth the gain. The principles laid out above however easily allow for incorporation of "black boxes" that expose an interface and adhere to well-defined mathematical properties, so that they can be reasoned over, but not changed. This concept will be developed further in the follow-up project to ProThOS.

ACKNOWLEDGMENT

This work was partially funded by the German Federal Ministry of Education and Research (BMBF) in the ProThOS project, Grant No. 01IH16011.

REFERENCES

[1] A. M. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. s2-42, no. 1, pp. 230–265, 1937.

[2] J. Hanby, "Software Maintenance: Understanding and Estimating Costs," 21-Oct-2016.

[3] Krugle Enterprise, "The Four Hidden Costs of Software Maintenance," 2014.

[4] K. Jeffery and L. Schubert, "Challenges in Software Engineering, H2020: Analysis and Summary of the Cloud Expert Group Reports," *European Commission*, 2014.

[5] L. Schubert and K. Jeffery, "Complete Computing: toward information, incentive and intention," *European Commission*, 2014.

[6] L. Schubert, A. Tsitsipas, and K. Jeffery, "Establishing a basis for new software engineering principles," *Internet of Things*, vol. 3–4, pp. 187–195, Oct. 2018.

[7] H. Nilsson and International Symposium on Trends in Functional Programming, Eds., *Trends in Functional Programming. Volume 7 Volume 7*. Bristol, UK; Chicago, IL: Intellect Books, 2007.

[8] I. Wegener, Complexity theory: exploring the limits of efficient algorithms. Berlin ; New York: Springer, 2005.

[9] P.-O. Ostberg *et al.*, "The CACTOS Vision of Context-Aware Cloud Topology Optimization and Simulation," 2014, pp. 26–31.

[10] B. O. Fagginger Auer and R. H. Bisseling, "A GPU Algorithm for Greedy Graph Matching," in *Facing the Multicore - Challenge II*, vol. 7174, R. Keller, D. Kramer, and J.-P. Weiss, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 108–119.

 ISBN: 978-1-61208-732-0