A Lower Bound Study on Software Development Effort

Lung-Lung Liu llliu@mail.mcu.edu.tw
International College Ming Chuan University
Gui-Shan, Taoyuan County,
Taiwan, ROC 333

Abstract

This paper depicts a study on the lower bound of software development effort. The work begins with the transformation model which is with the popular software development lifecycle. In general, a combination of properly handled transformations can ultimately produce the software, and the transformations form the path of software development. Each transformation is associated with its effort, or the weight, in a path. There can be a number of these paths since many different methods and tools can be used or reused for a specific software development case. Then, the Shortest Path algorithm is applied to find a shortest path which is with a minimal total effort among all the paths. However, from time to time, when advanced methods and tools are introduced, the new paths and efforts will change the previously identified shortest path. Hence, the continued work is to discuss the minimal total effort of a potential shortest path, although it may be currently unavailable. Convergence analysis is firstly provided for the discussion of whether this shortest path exists, and lower bound analysis is then provided for the discussion of completeness and soundness. This lower bound study is significant, since an optimal software development effort is determinable.

Keywords: Lower Bound, Software Development, Effort

1. INTRODUCTION

Software development effort has been a research topic since the early 1980's [1]. Most of the related studies are on the estimation of the effort, or the cost, of software development. In the review paper [2], it is indicated that there have been great contributions in this field. Some of them came from the academic, and some others came from the industry. In the other paper [3], a debate on "Formal Models or Expert Judgment?" is in discussion, and this is a typical example that to both researchers and practitioners, the effort issue is interesting. The estimated result is usually a predicted range of values (such as 200~250 person-months), and they can be obtained by applying probabilistic models and referencing historical data. For example, if a team has been working together in similar projects for five years, then their coming year's productivity can be easily predicted by applying a simple curve fitting approach. However, as team members are to change, project styles are to change, and working environments are to change, there are more and more factors to take into consideration. The probabilistic models assume that the factors are random variables with proper distribution functions and then perform the calculation. Although these models are just at the entry level to the studies, these seemed-to-be-straight descriptions may have blocked the software engineering practitioners to pay attention to the estimation fundamentals. They are complicate, and they are hard to be practiced.

In this paper, we are to depict a lower bound study on software development effort. It is with a computer science oriented algorithmic approach [4] but not the traditional probabilistic model approach. We begin with a transformation model which is based on the popular software development lifecycle and the programming language support systems. Successful and efficient transformations are mechanisms such as the compilers. They are able to change the external forms of software, while the internal computation essentials are still the same. Hence, when several transformations are performed, an executable image which is equivalent to an originally specified requirement is expected, and it can be loaded into a computer for a run. Every transformation is with an effort. If it is close to zero, then the transformation is efficient. On the other hand, if it is a relatively large value, then perhaps the project is to fail and alternatives should be considered. According to the transformations applied to a specific development case, a path through the transformations can be drawn. We also put the effort of each transformation as the value associated to the element in the path. Since there are many possible development cases, there are many paths. Now it is the Shortest Path problem, and the well-known Dijkstra's algorithm [5] can be used to find the shortest path. If we can prove that the shortest path does exist among these possible development cases, then there is the lower bound.

The proof is with the discussion on whether the approaches applied in a transformation is convergent. For example, iterations are frequently introduced in software development methods. However, without the proper control, the iterative processes may cause an infinite effort. A path with this transformation will never be the shortest path. On the other hand, since nowadays advanced software tools for efficient development methods are publically available, there are transformations that their efforts are close to zero. We want to identify these transformations and set them aside, and we are to check the remaining for whether further analysis is still doable. Theoretically, with a full set of reusable modules (which can be quite large) ready, the lower bound of software development effort can be with linear complexity, counted on the number of possible requirements specified. In the following, the transformation model is introduced in Section 2, and the applying of the Shortest Path algorithm is described in Section 3. The convergence analysis and lower bound analysis are provided in Sections 4 and 5, and the requirement specification that dominates the bound is discussed in Section 6. Finally, the conclusions are in Section 7.

2. THE TRANFORMATION MODEL

The set of integrated compile-link-load processes is a typical example for the transformation model. Let abc.c be a source program written in C language. With a C compiler for a specific development environment, abc.c can be transformed (compiled) into abc.obj, or an object program. With a link editor (or other equivalent tools), abc.obj can be transformed (added with other necessary object programs externally referenced) into abc.exe, which is a ready to run loadable module. Finally, with a loader, the load module is transformed (for relative address resolution, basically) into an image of a memory map and then copied into the memory for a real run. The above mentioned source program, object program, load module, and image are totally different in their formats, but actually they are the same from an essential software point of view. They are with exactly the same logical sequences of machine instructions that are for a desired computation goal. If they are not, then the language subsystem supported for that development environment is incomplete. The transformations are depicted in Figure 1.

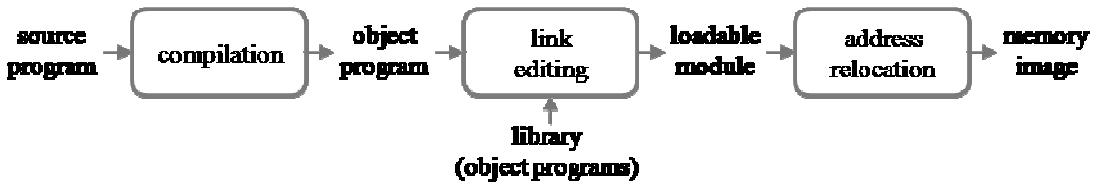

FIGURE 1: Program transformation

In the popular software development lifecycle [6], there are stages. The stages can be individually suggested, depending on the approaches or methodologies applied. For example, in a common Waterfall based approach, they may be the requirement analysis stage, the design stage, the implementation stage, and the maintenance stage. Continuing with the previous compile-link-load case, we only take the first three of the stages into consideration, since the last one is currently out of our scope. The associated transformations are to ultimately produce the source program abc.c (after that the language subsystem will properly handle the following details). In their natural sequences, the major functions of the transformations are to (1) transform the actual requirement in stakeholders' mind (that is in the cloud) into a requirement specification (that must be in machine readable format for further automation), (2) transform the requirement specification into a design specification, and (3) transform the design specification into a pseudo code, or equivalently, the source program. The transformations are depicted in Figure 2.

FIGURE 2: Specification transformation

We check the transformations one by one, starting from the last (which is the transformed source program) to the first, to see the difficulties. In transformation (3), current methods and tools can easily help the works, since there are polynomial time algorithms to solve tree traversal problems and grammar related parsing problems. The assumptions here are that the design specification has clearly indicated the number of modules, their interfaces to one another, and their computation basics with data structures, and that the programming language used is with strict syntax rules. The cases in transformation (2) are complicated. A key problem is that the common requirements specified are with a network structure, since they can be related to one another or they are independent. If the modules in the design specification are expected to be with a hierarchical structure (such that the programs can be with a perfect top-down tree structure), then we need an algorithm to convert a thing in network structure into some other thing in an equivalent tree structure. However, there is no such an algorithm. Software developers are used to taking a heuristic approach (with their experience) to try to find proper groups of requirements and let them be in a hierarchy. In some of the (worst) cases, the proper groups could never be found. On the other hand, if the modules in the design specification are not expected to be in any structure, or they are still in a network structure, then the works are easy but a non-procedural programming language support is necessary in the following development stags. In transformation (3), the actual work is requirement elicitation, which is to put outside information (in the stakeholders' mind, actually) into the computer as the first version of digitized (machine readable) data for further software development details. Before that, the computers can do nothing since there is no data to process. The transformation model is a basis for the study of software development effort, since ultimately programs are produced with a sequence of proper transformations performed.

3. APPLYING THE SHORTEST PATH ALGORITHM

Transformations are with efforts, and a sequence of transformations is with a total effort. Let e_i be the effort of performing transformation *i*. If in a graph the nodes represent the things before and after a transformation, and the edges (with direction) represent the effort of the transformation, then a sequence of transformation can be a *path* made of efforts associated. An example of the path made of transformations mentioned earlier is given in Figure 3.

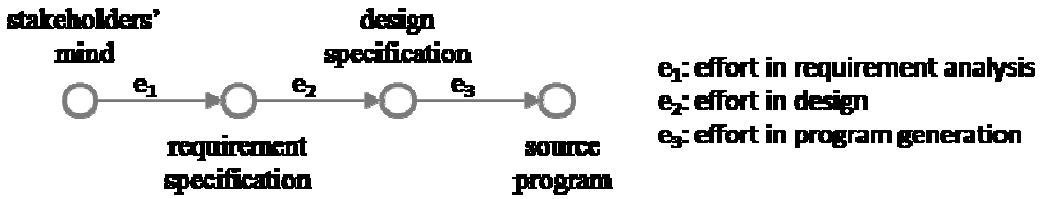

FIGURE 3: A path made of transformations

Occasionally, there are multiple paths, since there are different methods and tools that can be applied to a specific transformation. The individual efforts are different, and the total efforts are different. In addition, a transformation may be skipped when an advanced methodology is applied (such as an interpreter is used for a compiler in the language support subsystem). At this moment, we temporarily focus on the topic that multiple paths are there, and that why they are available is to be discussed in the next section. An example of multiple paths with a same sequence of transformations (same methodology used, but different methods used in a transformation) is provided in Figure 4, and another example of multiple paths with different sequences of transformations (different methodologies used, hence a transformation may be skipped) is provided in Figure 5. In the first example, there are three paths, with efforts of $e_{11}+e_{12}+e_{13}$, $e_{21}+e_{22}+e_{23}$, $e_{31}+e_{32}+e_{33}$, respectively. In the second example, there are four paths, with efforts of $e_1+e_2+e_3$, e_1+e_5, e_4+e_3, e_6, respectively. There may be more complicated multiple paths, since a combination of the cases provided in the example are all possible.

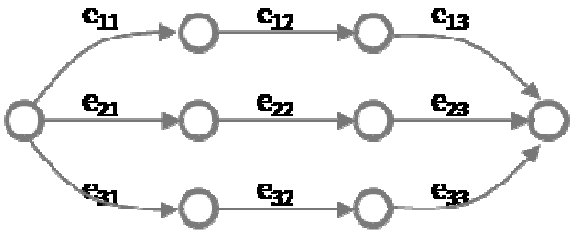

FIGURE 4: Multiple paths with a same sequence of transformations

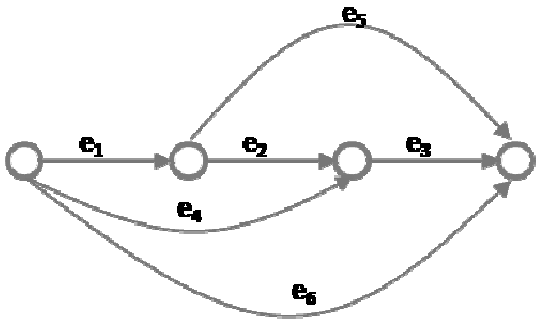

FIGURE 5: Multiple paths with different sequences of transformations

When paths are well defined, we are to find a path which is with the least effort among all of them. It is natural that Dijkstra's famous Shortest Path algorithm can be applied for a solution. The problem can be solved easily, but there are conditions. For example, if in every path there is a different effort which is with infinite as its value, then every path is with a total effort of infinite and there will be no shortest path at all.

4. CONVERGENCE ANALYSIS

The major purpose of doing convergence analysis is to make sure that the Shortest Path algorithm work well and hence is useful. There are at least two factors as conditions that should be taken into consideration: the possible values of every individual effort *e*, and the way the computation like "e_1+e_2" is to be performed. If both of the factors are with the convergence features, then the shortest path exists.

First, the value of an effort should be greater than zero. An effort with a negative value means that a transformation can even recover the resources (such as man-power, time, and money) exhausted before, but it is physically impossible since times elapse only. A zero value effort is a redundant transformation which should be removed from the associated path. However, an effort with a positive value close to zero may mean that the transformation has fully utilized the skills of tool automation, and it is encouraged. On the other hand, an effort with a relatively large may mean that the transformation is not efficient, and it is to be improved. The worst case is that the value is close to, say, infinite. Although it is only mathematically possible (infinite does not exist practically), it happened and project failed due to poor transformation (or no convergence management) works were performed.

The poor cases are popularly available, and from the convergence point of view, they really offend. There are typical examples, and we propose three of them. The iteration based methods are risky, since works may be performed repeatedly but with no progress. A criterion to stop the iteration should be set up in advance (even it is budget oriented), just like the defining of termination conditions in a recursive procedure. The introducing of the divide-and-conquer strategy may be another trap in some of the poor cases. The divide part makes the developer a significant progress, but, after the conquer part there always is a merge part which is rarely emphasized. When the effort of performing a merge is greater than that of solving the original problem, the convergence is broken. Besides, only independent works are suggested to be divided and conquered, otherwise the effort on synchronization is added in. The last one is related to the debugging work, and that happened to the junior programmers. Once an error in a program was found and is to be debugged, there is the chance that the error was corrected while new errors are added. The total number of errors is increasing but not decreasing, hence the transformation is not with convergence.

Secondly, the way the computation like "e_1+e_2" should be performed simply. For example, it is just a normal addition of numbers. There is no complicated definition on the operator "+" and there is no specific domain for the operands. To control the conditions related to this factor, the immediate consideration is that the transformations must be independent hence the computation of the joint efforts of two consecutive transformations can be straight. Otherwise, we have to clarify the dependency features and then see whether they can be calculated. The standards of CASE tools interconnections [7] are the best references for this. The standards define the common data formats between two consecutive transformations and the disciplines to access the data. Following the standards to apply the methods and tools in the transformations will guarantee a smooth and costless interconnection. This also means that there is no extra effort on data conversion, and the "+" operation is the simplest arithmetical addition.

5. LOWER BOUND ANALYSIS

The purpose of providing the lower bound analysis is to formally prove the logical completeness and soundness in this study. In the following paragraphs, Facts and Theorems are given. The Facts are always true, and they are with explanations only. Theorems are with proofs.

Fact 1. The transformation model works.

That is, the belief of software development lifecycle is feasible by applying the model. Let T be the set of transformations, and let t in T be a transformation that takes any d in domain D and then transforms d into a specific r in range R. In short, $t : D \rightarrow R$, and $r = t(d)$. If both t_i and t_j are in T, then the result of consecutive transformations t_i and t_j performed to d_i is $t_j(t_i(d_i))$. A sequence of transformations, such as the requirement analysis, design, and implementation stages suggested in the conventional software development approaches, really ultimately generate the source programs. We leave the definitions of the sets T, D, R free.

Fact 2. A path representing the sequence and efforts of transformations performed in a specific software development case is determinable.

That is, a path is also with the transformation model but in a different format used in graphs. Let $r = t(d)$ be with the definitions used previously. Equivalently, t can be represented by using an edge e connecting nodes d and r, with a direction, and the value associated with e is the effort of performing transformation t. Then, a sequence of n transformations t_1, t_2, t_3, …, t_n firstly performed to d is a path $p : e_1 \rightarrow e_2 \rightarrow e_3 \rightarrow \ldots \rightarrow e_n$, started at d and ended at the lastly generated (what was transformed into) specific r. The total effort of the path p can be derived from e_1, e_2, e_3, …, e_n.

Fact 3. The shortest path from node a to node b represents the least effort among all possible software development cases targeted for a specific application, given a as the initial input and b as the final output.

That is, the least effort can be determined by finding the shortest path among all the possible paths derived from those cases. In other words, within two different paths, if they have common joint nodes (different transformations were performed for a common output, and then other different transformations continued), then there can be a combined new path which is with the first half sequence of one path and the second half sequence of the other. However, the new path is with less effort than the previous two. Let $p_1 : e_1 \rightarrow e_2$ and $p_2 : e_3 \rightarrow e_4$ be the two paths. The case happens when there is a joint node at the middle of these two paths, with the condition that $e_1 + e_4 < e_1 + e_2$ and $e_1 + e_4 < e_3 + e_4$.

Theorem 1. The Shortest Path algorithm can be applied to find the least effort of software development cases to a specific application.

Proof: According to Fact 1, the transformation model is applicable to software development processes, starting from the requirement specification stage for the source programs. According to Fact 2, the path representing a sequence of transformations can be properly defined in a graph, with transformation efforts as weights of the edges. According to Fact 3, the shortest path is the sequence of transformations with the least software development effort among all the possible cases. Let $G = \{V, E\}$, where V is the set of nodes and E is the set of edges with weights representing the transformation efforts, be the graph derived from the software development cases to a specific application. Since the Shortest Path algorithm is bounded by the number of nodes V in the graph (the computation time is of order V^2 or $V \log V$), the finding of the shortest path is guaranteed. The total weight, cumulated by those associated to the edges forming the path, is the least effort of the software development cases.

Theorem 2. The lower bound of software development effort can be found if all the development cases are collected.

Proof: Let there be n possible paths according to all the cases. By Theorem 1, the Shortest Path algorithm can be applied to find the least effort e_s. That is, $e_s < e_i$, i=1, n, i ≠ s. The lower bound L is e_s, since for every e_i, i=1, n, L ≤ e_i.

Fact 4. The effort of using tools is close to zero.

That is, when tools are used, there is almost no effort, compared with the barehanded way to do the development work. Here the effort is only on the using but not the introducing of tools. Tools are software running on computers, and the performance is faster than human in millions of times. We can conclude that the effort of using tools is close to zero.

Theorem 3. By fully applying the tools, the lower bound of software development effort is with a linear complexity, counted on the number of possible requirement specified.

Proof: Let there be powerful tools and software development environment supports, although some of them are currently unavailable. Let the shortest path found be made of only one edge, and it is associated with the least effort among all edges. This has actually indicated an efficient and effective methodology that is able to directly transform the requirement specified into a source program. Since the effort of using tools is close to zero according to Fact 4, we neglect them. Then the effort is to let the tools be able to recognize the possible requirements. Let the requirements be specified in a yes/no feature list, and let the number of the features in the list be *n*. The effort of handling all the possible requirements specified is with a linear complexity of *n*.

6. REQUIREMENT SPECIFICATION IN SELECTION

Works related to requirement analysis are actually with the most effort in software development [8], since the works at other development stages have almost been fully automated by using tools or even skipped. For example, there are very high level requirement specification languages and the support systems that can accept requirements as statements in a special syntax, such as the Prolog programming language and her runtime environment support. In Prolog, a requirement can be represented by a number of logical rules, and an equivalent number of statements directly translated from the rules actually completed the source program, or the software development works. The other example is the object oriented development methods. Usually, at the end of object oriented analysis stage, use cases should have been generated, and then at the design stage, all the necessary objects are to be determined. If the class libraries are well classified and indexed, then that will be just a mapping work. (However, the classification problems or the domain analysis issues are beyond the discussion scope here.) These two examples have indicated that the effort for transformation from a dataflow diagram to a structured chart in the conventional design stage could be totally skipped.

The focus has been on the minimization of the scope that developers should really pay attention to, and it is in the handling of requirement analysis. There are cases that, for some specific requirements, what actually in the stakeholders' minds are unclear, even they themselves cannot tell. Conventionally, there are at least three necessary steps in the requirement analysis stage. The first one is requirement elicitation, the second one is requirement integration, and the last one is requirement negotiation. It is a discipline that requirements ought to be specified as items (such as use cases), and they are countable by numbers. Then, the three steps can be more precise. Requirement elicitation means a result of identifiable items shown in a list or a network structure. Requirement integration means a result of being integrity among the items. Requirement negotiation means a result of practical (developable) items after conflict resolution is done. The key is, the numbers of items mentioned above are all countable.

If there is another discipline says that all the requirement items specified should be flat, or with no hierarchy among them, then potentially the tools can do more. Let there be a rich reusable library of software components, supported by an intelligent configuration mechanism. A tool is suggested to ask *n* yes/no questions about a targeted application, and answers are expected. The number of the combination of all the possible answers is 2^n. If the tool is able to provide all of the source programs according to all of the possible answer sets, then the software development is done. Although the number 2^n is of exponential, practically when *n* can be controlled in an affordable range, the providing of such tools is considerable. In this way, the only effort in requirement

analysis stage is to do requirements specification in selection. It is to make decisions on the selecting of yes/no answers to the *n* questions. The complexity of the effort is of linear to *n*.

7. CONCLUSIONS

We briefly (1) specified the transformation model in software development, (2) mapped the transformations into a graph where a path representing the total efforts of a sequence of transformations, and (3) suggested that the Shortest Path algorithm can be applied to find the shortest path (if it exists) which represents the least effort among all the development cases to a specific application. We also provided the convergence analysis to discuss the possible infinite efforts due to iterative but not effective processes performed in the transformations, because if the convergence management is not considered, then the shortest path may not exist. Finally, we provided the lower bound analysis to show the logical completeness and soundness of the whole study. We are to conclude that once all of the possible development cases (including the currently unavailable ones) are checked, the shortest path found among the mapped paths is the one with least effort, and this least effort is the lower bound.

Current results indicated that the lower bound can be with a linear complexity, counted on the number of requirements specified. The assumption is that the reusable modules and the configuration mechanisms are powerful enough, just like the customers are to configure an operating system on their computers. They can obtain the custom-made software by simply setting the parameters provided in a customizable profile. The configuration tools will provide a combination of modules that fits the requirements of the desired operating system. The original software development problem is with a complicated complexity, but the tools may have solved most of them. For example, they absorb the ones with exponential complexity, while leave the one with linear complexity to the developers. However, these are the currently unavailable ones.

The future works include (1) the analysis of the conditions that may hold a minimum effort such that the lower bound is thus blocked, (2) the analysis of the cases that methods in transformations did not really reduce the effort but just postpone the works, and (3) the discussion for possible further formal proofs.

8. REFERENCES

[1] R. W. Selby and B. W. Boehm. *"Software Engineering: Barry W. Boehm's Lifetime Contributions to Software Development, Management, and Research".* Wiley-IEEE, 2007

[2] M. Jørgensen, M. Sheppered. *"A Systematic Review of Software Development Cost Estimation Studies".* IEEE Transactions on Software Engineering, 33(1):33-53, 2007

[3] M. Jørgensen, B. Boehm, S. Rifkin. *"Software Development Effort Estimation: Formal Models or Expert Judgment?"* IEEE Software, 26(2):14-19, 2009

[4] D. E. Knuth. *"The Art of Computer Programming, Volume 4, Fascicle 0: Introduction to Combinatorial Algorithms and Boolean Functions".* Addison-Wesley Professional, 2008

[5] Dijkstra, E. W. *"A Note on Two Problems in Connexion with Graphs, in Numerische Mathematik, Vol. 1".* Mathematisch Centrum, Amsterdam, the Netherlands, pp. 269-271, 1959

[6] ISO/IEC 12207. *"Information Technology – Software Lifecycle Processes".* IEEE/IEC Standard 12207.

[7] *"IEEE Guide for CASE Tool Interconnections – Classification and Description (IEEE std 1175)".* IEEE Standards, Available at: http://ieeexplore.ieee.org/.

[8] J. Lee and N. L. Xue. *"Analyzing User Requirements by Use Cases: A Goal-Driven Approach"*. IEEE Software, 1999