Two frameworks for task analysis: An application to the use of intelligent systems in the development of information systems

Pertti H. Järvinen

University of Tampere

P.O.Box 607

FIN-33101 Tampere, Finland

phone +358-31-2156 777 fax +358-31-2156 070 e-mail pj@cs.uta.fi

Abstract

The productivity paradox of information technology has said to be one of the most difficult problem of computing field. We must try to improve productivity of both computing applications and software industry. It is therefore vital to identify and differentiate productive efforts from non-productive ones.

In this paper we develop two frameworks for analysis of tasks in the development of information systems. When we automate part of the development, we at the same moment distribute tasks between man and computer. We shall show that this distribution will create some non-productive additional subtasks: Transportation, communication, inspection, coordination, coding/encoding and conversion. Our second framework consists of the initialisation and termination subtasks surrounded every process proper in a discontinuous, non-repetitive task.

By making those implicit subtasks explicit and demonstrating their existence both in generative and building blocks approaches we want to increase our awareness, to improve productivity and to move consideration and management of those subtasks under conscious control.

Keywords

Computer-aided software engineering, programming environments, productivity, distribution of tasks

1 INTRODUCTION

Brynjolfson (1993) referred to the study indicating that output per production worker grew by 16.9% between the mid-1970s and 1986, while output per information worker decreased by 6.6 %. Although he gave four explanations for the productivity paradox, we must not be pleased with the situation. Software productivity has been steadily increasing over the past 30 years, but not enough (Boehm 1987). Despite of intensive research, the development of information systems (IS) under acceptable conditions of quality and productivity has not fulfilled promises. A great deal of progress has been achieved in understanding the process of developing a program from a specification, little progress has been achieved in improving the practice of information systems development (ISD) accordingly. This reflects the situation that the main emphasis in software construction research has been in minute details about semantics of programming languages and correctness formulas but not in productivity and

quality. According to Mili et al. (1995) automated tools (here called intelligent systems) may at least partially solve the problem above. They suspect that the choice of algorithms, control structures, and data structures are difficult to formalize and hence to automate.

In this paper we concentrate our efforts on the development of MIS applications where intelligent systems are used. One of the main problems in this field has been an unsatisfactory co-operation between users and systems analysts. In this study we exclude social problems sometimes occurring between those two groups. The distribution of tasks to two groups still seems to cause some other additional subtasks. For example, according to what systems analysts have experienced, their derivation of specifications from requirements prepared by users has not been as simple nor as straightforward as expected.

The use of an intelligent system in development, in fact, means distribution of tasks between man and machine, and the similar additional subtasks may be found in this case, too. In the first part of this article we are interested in to explicate those additional subtasks, to show that they are non-productive and to form a framework. In the second part we apply our first framework to various alternative cases where intelligent systems are used.

In addition to distribution of tasks between two processors, a job of a particular processor is not always composed of one task or the repetition of it only, but often of many different tasks. In the former case we can speak about performing the continuous process, and in the latter case the discontinuous processes. In the job with discontinuous tasks, the developer must move from the earlier task to a new different task. The developer must firstly free him/herself from the earlier task by performing some termination activities. The developer must then perform some initialisation efforts for the new task before accomplishing the task proper. In the first part of the paper we also describe our second framework (initialisation-termination) and shall in the second part apply it to different development alternatives.

To describe our object of study and selection of a methodological approach we inform that we shall in our theoretical analysis restrict our efforts in rationalisation of the information systems development, especially in distribution of tasks, and hence we exclude social, political, psychological etc. aspects of division of labour. In consideration of distribution of tasks we particularly try to differentiate productive efforts from non-productive ones.

To structure the second part of the paper we firstly refer to the fact that in the software reuse work (Horowitz and Munson 1984) it is customary to differentiate two approaches: 1. The method reuse, which is based on reusing the process of previous software development efforts, often embodied in tools that automate part of the development, called *generative approach*, and 2. the *building blocks approach*, which is based on reusing software development products (Krueger 1992). This order, some generative approaches and then building blocks approach, is followed in the second part of the paper.

2 Part 1. TWO THEORETICAL FRAMEWORKS ON DISTRIBUTION OF TASKS BETWEEN TWO PROCESSORS

2.1 On distribution of tasks between two processors

In this section we shall outline a conceptual framework intended to an analysis of the intelligent systems used in the ISD. By *intelligent systems* we mean the computing systems supporting the development, inauguration, use and maintenance of IT systems either separately or together. The purpose to use some intelligent system is based on an assumption that this kind of system can support and complement human capabilities in such a way that the use is forecasted to be profitable, i.e. the construction of the IS under development will be completed faster or will cost less than by using other means.

An essential fact in using intelligent systems in the ISD is that the development tasks are then distributed between man and computer. We must therefore consider potential consequences caused by this distribution of tasks. We shall show that the distribution seems

always cause some new additional subtasks and the consideration below will deepen our earlier sketch (Järvinen 1992). We try to keep our consideration as general as possible.

The performance of a task is here considered as transformation of an object of work (in our case: data) from one form to another, e.g. from E_i to E_j , and from E_k to E_f . If two processors, say P1 and P2, have different locations, the outcome E_j after completion of task Ω must be transported to P2 (otherwise P1 must move from and P2 to the place where E_j exists). In order to guarantee a continuation of work from one processor to the other, some *communication* between P1 and P2 is needed. P1 must inform P2 of E_j and P2 must give some feedback to P1 after receiving $E_j=E_k$. The two processors are responsible for their outcomes: P1 is responsible for E_j and P2 for E_f , respectively. The shared responsibility makes necessary *two inspections* concerning $E_j=E_k$: after task Ω and before task π . It is checked against predetermined standards. To prevent delay in delivery of $E_j=E_k$ to P2 on one hand and to eliminate idle time of P2 on the other hand, and to promote the harmonious cooperation of P1 and P2 some *coordination* is needed. Our analysis shows that the distribution of two consecutive tasks to different processors create some additional subtasks: Transportation, communication, inspection and coordination (Figure 1).

Figure 1 Some non-productive subtasks caused by distribution of tasks.

Those identified new subtasks (transportation, communication, inspection and coordination) are *non-productive* in the sense that their performance does not contribute the completion of the final outcome E_f . Those non-productive additional subtasks could be avoided, if Ω and π had the same processor.

The use of man and computer as adjacent processors may cause some further subtasks. The presentation format of data for man and for computer can be different. This causes a demand for *conversion* from one representation to the other. Sometimes an attempt is made to minimize the amount of characters to be transported by *coding* (decoding).

2.2 On discontinuous characteristics of processes

In the operations research literature (Conway, Maxwell and Miller, 1967) it has first time been paid attention to effects caused by some necessary set up activities required before the productive use of some machine. I reviewed and identified some use of initialisation and termination tasks surrounding the process proper in program design (Järvinen 1987).

In program construction we have found that in the beginning of the program some preparations are required before the process proper, the application of the main algorithm, i.e. *initialisation* of counters, sum fields and different flags, opening files etc. Correspondingly, after the execution of the main algorithm some *termination* activities, i.e. printing the results, closing the files, etc. are necessary (Figure 2). The differentiation between initialisation and

termination on one hand and the process proper on the other hand means differentiation between non-productive and productive tasks.

Figure 2 Initialisation (i) and termination (f) activities surrounding task T.

The initialisation and termination activities required before and after repetitive sub process for the similar transactions are considerably less than those needed in the beginning and at the end of the whole program (lower part of Figure 3).

Figure 3 Initialisation (i) and termination (f) in dis- and continuous processes.

Next we apply our two frameworks to the information systems development process. We shall firstly analyse some forms of a generative approach and finally the building blocks approach.

3 Part 2. APPLICATION OF TWO FRAMEWORKS FOR THE DEVELOPMENT

3.1 Some generative approaches

The support of intelligent systems can take place in different settings. Many CASE tools support in various phases of the ISD, but between those automated periods some manual efforts are needed. There are intelligent systems that support the rest of the development process, starting from a certain phase until to the end. The latter can takes place as one task, e.g. when an interpreter is used, or in several transformation phases. We shall firstly consider those three alternatives separately: 1. to support in various phases, 2. to support several transformations and 3. to support the rest of the process.

To support in various phases (alternative 1)

Supporting the information systems development (ISD) by intelligent systems is not the new idea, for the ISDOS (Teichroew and Sayani, 1971) and the SCIP projects (Bubenko, Langefors and Sölvberg, 1971) generated programs to support documentation, design and code generation. Concerning the whole development process those intelligent systems only here and there supported the completion of an application system. The CASE tools offer such different approaches to the analysis and design process that the prospective user should decide on his requirements and underlying methodology, before determining the best match

with the tool (Crozier et al. 1989). In the special issue of CASE tools Norman and Forte (1992) presented one item as follows: "Shepard, Sibbald, and Wortley's 'A Visual Software Process Language' asserts that next-generation CASE environments need ability to create a customized model of the development process used. This model will assist in the automation of some portions of the process, guide others, and remove an administrative burden from developers."

Winograd (1995) recently proposed that the environment for the designer goes well beyond the traditional bounds of programming environments. He wanted to expand current interactive programming to responsive prototyping media at four prototyping levels: (i) Rough hand sketches and scenarios, (ii) low-fidelity prototypes (wizard of oz), (iii) programmed facades and (iv) prototype-oriented languages. According to his view the full design environment is a mix-and-match of all of these prototyping levels. To our mind, Winograd has not argued the questions: How to move from one level of prototyping to another level? How many times a processor must be changed?

Mili et al. (1995) gave some formulae to calculate software development costs, when the building blocks approach was considered, but they did not give the similar formulae for generative approach. Misra and Jalics (1988) and Verner and Tate (1988) gave some crude guidelines only, but they did not pay attention the use of fourth-generation languages (as intelligent systems) in the automation of some portions of the development process. Our first framework shows that change of processors (between man and machine) creates all those additional subtasks (alternative 1):

- Transportation: developers must input their specifications, descriptions of data and algorithms, etc.; the intelligent system must output results of the process performed.
- Communication: developers must inform the intelligent system that it must perform a certain process, and after execution or in possible breakdown situations the intelligent system must inform developers.
- Inspections: developers must check that input to be given is correct, and the intelligent system must ascertain that the matter is so.
- Coordination: developers must use the intelligent system in the suitable phase in the development, and the system must examine that all the necessary data and guiding parameters are available before execution.
- Coding/encoding and conversion: the intelligent system requires input data in a certain form; developers must perform coding needed for input, and the system normally execute the needed conversion for both input and output.

We do not have any quantitative data about durability of those subtasks. The times can be short compared with productive ISD tasks, but it is important that they exist and we, computing professionals, perceive them and try to avoid them, because they are non-productive.

Our second framework, the initialisation-termination, says that two subtasks surround every process proper:

Initialisation: concerning the intelligent system 'begin' and 'end' with non-empty content are needed at the beginning and at the end of each program (Crockett, 1981); concerning a developer s/he must collect material, arrange a meeting with users, prepare interviews, purchase a software needed, start up the program etc. before the process proper.

Termination: in case of the program we refer to arguments above; concerning a developer's activities after the process proper s/he must file the documents, finish the run of the program, log out from the system, etc.

We have here only crudely described possible content of the initialisation and termination. We do not have an exhaustive list in connection with this nor with the previous framework.

To support several transformations (alternative 2)

Goos (1975) presented the principle of an abstract machine. When it is applied to the ISD, the application system can be automatically produced from a certain phase of the development to the end without human intervention. Goos pictorially described many phases, hence we can

say that the abstract machine principle also allows two or more compilations, i.e. the first specifications written with language A are first compiled with pre-processor A to the format with expressions of the B language, and the B format is then compiled with another processor or compiler, say B, to the form C, and the similar compilation process can continue until the object code version is achieved. The termination of the first compilation and the initialisation of the latter compilation are needed, because two compilers are used. Both subprocesses are automated and their procedures are simpler than in alternative 1. (There are some other interesting aspects presented in the longer version of this paper (Järvinen 1996).

To support the rest of the process (alternative 3)

The extreme situation is that the specifications as such are executable (Nota and Pacini, 1992; Sitaram and Shaw, 1994; Mili et al., 1995). This approach can be called one compilation or interpretation alternative in the sense that one compilation only is needed from the specifications or designs to the final outcome, i.e. to the system in the object code form. In this approach distribution of tasks could take place between man and machine only once, hence many additional non-productive subtasks found in various phases or between several transformations were now eliminated. To formulate and to feed in the specifications means the transfer of responsibility from man to machine, and in this point we can again find transportation, two inspections, communication and coordination. The descriptions made in connection with alternative 1 are valid here.

Our analysis contribute such a view that we should strive towards executable specifications and use fourth-generation languages. But Misra and Jalics (1988) and Verner and Tate (1988) compatibly show that it is sometimes difficult to present complex algorithms by using fourth-generation languages. Execution or interpretation times of applications made by fourth-generation languages are still slower than the ones in applications made by third-generation languages.

3.2 The building blocks approach

In the building blocks approach ready-made software components are reused. The development of an 'old' component has taken place long ago before its reuse. Concerning the life cycle of that component its reuse is a discontinuous event.

With the building blocks approach, developers try to build a system that satisfies a set of requirements by using as many existing components (or developing as little code) as possible. For any part of target system, developers must (Mili et al. 1995):

- (I) formulate the requirements of the part in a way that supports retrieval of potentially useful reusable components,
- (II) understand the retrieved components and
- (III) if the retrieved components are sufficiently "close" to the needs at hand and are of sufficient quality, then to adapt them.

If no component is found that matches perfectly or closely the given requirements, developers must return to apply a general problem-reduction heuristics to decompose the system (or part thereof) into smaller parts for which steps I ... III may be reiterated.

A general comment: If a developer knows the suitable component, s/he can immediately take it to the software under development. This is not very probable, because the components are prepared earlier and often by a different person. In this discontinuous case, retrieval from component library is a typical alternative. If the intelligent system is used for searching and retrieving the component from the component library, many initialisation activities are needed: encoding and formulating descriptions of the requirements into such a form (query) that comparison against encoded descriptions of the components in the library is possible.

The building blocks approach consists of three phases: 1. the top-down problem reduction phase, 2. component adaptations and 3. the bottom-up composition phase. We in this section structure our analysis according to those phases.

The top-down problem reduction means the recursive application of steps I ... III until some potential component is found for each leaf of the solution tree. This phase resembles alternative 1, because developers alternatively perform subproblem reduction and use the searching program. The results derived above are valid here, too.

The component adaptation is very easy, if it perfectly matches the requirements. A developer then move it for composition. This means transportation. If the component found does not perfectly match the requirements, it must be modified for the purposes of the desired IS system. We must keep this modification as a productive task, because it contribute the final outcome. The modification requires efforts both from developers and from an intelligent system. This situation again resembles alternative 1.

The bottom-up composition means that for a given set of requirements, a set of components within a component library whose combined behavior satisfies the requirements should be found. Mili et al. (1995) see this synthesizing perhaps the most challenging problem in the building blocks approach, and where computer assistance is much needed. This phase, thus, usually needs co-operation between developers and intelligent systems. One of the main tasks is to link one component with another, and to check whether the combination satisfies the requirements stated. Linking is at least productive task, but checking can be kept as inspection subtask caused by the methodology used. In general, this situation also resembles alternative 1.

On the library of components

The reason, we separately considered the building blocks approach, is the storing and retrieving functions applied to the component. As we mentioned above storing and retrieving bring the discontinuous nature for the building blocks approach, and hence more initialisation and termination subtasks than usually. Justification for the component library comes from the increased value of the ready-made component compared with the development from scratch. This reasoning leads to the consideration of *value-chains* (Porter, 1985) in software industry.

The modification in the component adaptation phase may produce such a version of the original component that the intelligent system can replace the original one in the library or store it as a new component. Some indexing and encoding or at least modification of descriptions of the library component is then needed. Those activities made for improving component library can be kept as investments from the perspective of the software house. The value-chain analysis may help to decide whether storing the new component into the library is profitable or not.

4 DISCUSSION

Our theoretical analysis firstly gave the framework: The distribution of two consecutive tasks to two different processors will create the non-productive additional subtasks: Transportation, communication, inspection, coordination, and possibly coding/encoding and conversion. Our second framework consists of the initialisation and termination subtasks surrounded every process proper in a discontinuous, non-repetitive task.

In thinking over practical consequences of our findings, we recommend to avoid division of labour by lengthening the sequence of consecutive tasks assigned to one processor. This will eliminate both non-productive additional subtasks, and at the same time diminish the number of initialisations and terminations. - We want, however, to remark that we are not totally against the division of labour. It clearly has its merits in many situations: 1) only one processor is capable and competent to perform a certain task, 2) a certain processor is superior in performing the task compared with other processors, and 3) assigning a new task to a particular processor would exceed the processor's capacity. But outside of situations 1-3 above our recommendation is applicable.

By referring to Kling and Jewett (1995) we can see many new potential opportunities, if instead of our rational analysis the naturalistic and open systems approach would be applied.

The new construct, a value chain, in connection with software development emerged and needs further research.

REFERENCES

- Boehm B. (1987) Improving software productivity. *IEEE Computer* **20**, No 9, 43-57.
- Brynjolfson E. (1993), The productivity paradox of information technology. *Comm. ACM* **36**, No 12, 67-77.
- Bubenko J., Langefors B. and Sølvsberg A. (Eds.) (1971) *Computer-aided information systems analysis and design*. Studentlitteratur, Lund.
- Conway R.W., Maxwell W.L. and Miller L.W. (1967) *Theory of scheduling*. Addison-Wesley, Reading Mass.
- Crockett D.W. (1981) Triform programs. *Comm. ACM* **24**, No. 6, 344-350.
- Crozier M., Glass D., Hughes J.G., Johnston W. and McChesney I. (1989) Critical analysis of tools for computer-aided software engineering. *Information and software technology* **31**, No 9, 486-496.
- Goos G. (1975) Hierarchies, in *Software engineering - An advanced course* (ed. Bauer), Springer-Verlag, Berlin, 29-46.
- Horowitz E. and Munson J.B. (1984) An expansive view of reusable software. *IEEE Transactions on Software Engineering* **10**, No 5, 477-487.
- Järvinen P. (1987) On initialization and exitialization in program design, in *Empirical Foundations of Information and Software Science III* (eds. J. Rasmussen and P. Zunde), Plenum, New York, 135-144.
- Järvinen P. (1992) Impact of electronic markets on work, in *Education and Society* (ed. Aiken), Proceedings of Information Processing 92, Vol. II, Elsevier Science Publishers, Amsterdam, 419-425.
- Järvinen P. (1996) The longer version of this paper is available as [/pub/reports/Canberra.ps.Z](#) via anonymous ftp from cs.uta.fi.
- Kling R. and Jewett T. (1995) The social design of worklife with computers and networks: An open natural systems perspective, in *Advances in Computers* (ed. Yovits), Academic Press, Orlando, 50 p
- Krueger C.W. (1992) Software reuse. *Computing Surveys* **24**, No 2, 131-183.
- Mili H., Mili F. and Mili A. (1995) Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering* **21**, No. 6, 528-562.
- Misra S.K. and Jalics P.J. (1988) Third-generation versus fourth-generation software development. *IEEE Software* **5**, No 4, 8-14.
- Norman R.J. and Forte G. (1992) Automating the software development process - CASE in the '90s. *Comm. ACM* **35**, No 4, 27.
- Nota G. and Pacini G. (1992) Querying of executable software specifications. *IEEE Transactions on Software Engineering* **18**, No. 8, 705-716.
- Porter M.E. (1985) *Competitive advantage*. Free Press, New York.
- Sitaram C.V.R. and Shaw A.C. (1994) A prototyping environment for specifying, executing and checking communicating real-time state machines. *Software-Practice and Experience* **24**, No. 2, 175-195.
- Teichroew D. and Sayani H. (1971) Automation of system building. *Datamation* **17**, No 16, 25-30.
- Verner J. and Tate G. (1988) Estimating size and effort in fourth-generation development. *IEEE Software* **5**, No 4, 15-22.
- Winograd T. (1995) From programming environments to environments for designing. *Comm. ACM* **38**, No 6, 65-74.