Computers in Industry 67 (2015) 86–96

Contents lists available at ScienceDirect

Computers in Industry

journal homepage: www.elsevier.com/locate/compind

Using extended Axiomatic Design theory to reduce complexities in Global Software Development projects

Hadi Kandjani [a,b], Madjid Tavana [c,d,*], Peter Bernus [b], Lian Wen [e], Amir Mohtarami [f]

[a] Department of Information Technology Management, Shahid Beheshti University, Tehran, Iran
[b] Centre for Enterprise Architecture Research and Management (CEARM), School of Information and Communication Technology, Griffith University, Brisbane, Australia
[c] Business Systems and Analytics Department, Lindback Distinguished Chair of Information Systems and Decision Sciences, La Salle University, Philadelphia, PA 19141, United States
[d] Business Information Systems Department, Faculty of Business Administration and Economics, University of Paderborn, D-33098 Paderborn, Germany
[e] Institute for Integrated and Intelligent Systems (IIIS), School of ICT, Griffith University, Brisbane, Australia
[f] Department of Information Technology Management, Faculty of Management and Economics, Tarbiat Modares University, Tehran, Iran

ARTICLE INFO

Article history:
Received 28 December 2013
Received in revised form 28 August 2014
Accepted 21 October 2014
Available online 3 December 2014

Keywords:
Global Software Development
Complexity
Extended Axiomatic Design theory
Kolmogorov complexity

ABSTRACT

Global Software Development (GSD) projects could be best understood as intrinsically complex adaptive living systems: they cannot purely be considered as 'designed systems', as deliberate design/control episodes and processes (using 'software engineering' models) are intermixed with emergent change episodes and processes (that may perhaps be explained by models). Therefore to understand GSD projects as complex systems we need to combine the state-of-the-art of GSD research, as addressed in the software engineering discipline, with results of other disciplines that study complexity (e.g. Enterprise Architecture, Complexity and Information Theory, Axiomatic Design theory). In this paper we study the complexity of GSD projects and propose an upper bound estimation of Kolmogorov complexity (KC) to estimate the information content (as a complexity measure) of project plans. We demonstrate using two hypothetical examples how good and bad project plans compare with respect to complexity, and propose the application of extended Axiomatic Design (AD) theory to reduce the complexity of GSD projects in the project planning stage, as well as to keep this complexity as low as possible during the project execution stage.

1. Introduction

A Global Software Development (GSD) project has to go through complex processes to finish projects within an allocated budget, time schedule, and with customer satisfaction and completely fulfilled functional and non-functional requirements. The concept of GSD implies distributed teams from different organisations and geographical locations who collaborate to design, manage and execute life cycle activities of a joint software development project functioning as a supply chain [32]. This structure in itself increases the complexity of distributed GSD projects [13,35,38], where part of this complexity is due to dynamic dependencies among components of the software development products [7] as well as dependencies among life cycle activities of project planning and software development groups [1]. This complexity creates uncertainty and ambiguity due to the high number of elements and also the high amount of dependencies among GSD products, projects or project activities [29].

Given the highly distributed nature of GSD projects a completely centralised control is very hard to achieve, and subsequently these projects could be looked at as intrinsically complex adaptive systems: they cannot purely be considered as 'designed systems', as deliberate design/control episodes and processes ('software engineering', using models) are intermixed with emergent change episodes and processes (that may perhaps be explained by models).

There exist various kinds of engineered systems, including software products, which are developed by a global engineering effort. Common to all is a highly complex (or complicated) project

* Corresponding author at: Business Systems and Analytics Department, Lindback Distinguished Chair of Information Systems and Decision Sciences, La Salle University, Philadelphia, PA 19141, United States. Tel.: +1 215 951 1129; fax: +1 267 295 2854.
E-mail addresses: h.kandjani@griffith.edu.au (H. Kandjani), tavana@lasalle.edu (M. Tavana), p.bernus@griffith.edu.au (P. Bernus), l.wen@griffith.edu.au (L. Wen), A.Mohtarami@modares.ac.ir (A. Mohtarami).
URL: http://tavana.us/ (M. Tavana),

http://dx.doi.org/10.1016/j.compind.2014.10.008
0166-3615/© 2014 Elsevier B.V. All rights reserved.

design, as many of these projects have been usually designed "without having a theoretical framework for complexity" [43]. GSD therefore is becoming more complicated unless fundamental theories and principles, and corresponding methods for reducing complexity are developed (or adopted from the complexity field). An ultimate goal of the complexity field is to replace the "empirical approach" in designing, operating and managing complex systems with a more "scientific approach" [43]. Complexity is therefore an important problem facing GSD projects, because uncontrolled complexity can cause undesired design qualities and therefore unsatisfied requirements of GSD projects. The first question that may arise, before going any further, is: "What is Complexity?"

Gershenson [15] defines the complexity of a system (C_{sys}) as a function of the number of its elements ($\#E$), the number of interactions between them ($\#I$), the complexities of the elements (C_{ej}), and the complexities of the interactions (C_{ik}) among elements. Axiomatic Design (AD) theory [42] defines a 'complex' system as one that cannot be predicted to always satisfy its functional requirements. Suh [42] and other authors, such as Melvin [30], define the concept of system complexity through considering 'the probability of satisfying all functional requirements all the time'. Functional requirements are defined in AD as "a minimum set of independent requirements that completely characterise the functional needs of a product (software, organisation, systems, etc.) in the function domain" [39,42].

For software engineers the notion of a software not always satisfying its functional requirements may seem odd, a normal reaction to such state of affairs would be that this is due to the lack of a complete verification. However, in large scale systems verification cannot be complete, especially because one must take into account that the ability to produce the correct output by transforming an input that satisfies the preconditions, depends on other 'assumed inputs', for example that at the time the transformation must take place, the necessary processing power and storage are available. Even if every component of a system was designed to perform perfectly in isolation, they would not necessarily always perform accordingly as part of a system in every possible operational scenario (with a potentially intractable number of possible operational states), implying the need for a design theory that explains, and for methods that can be used to reduce, the complexity of a system.

Axiomatic Design is a theory that aims to distil into two 'design axioms' the essence of what is a good design, especially from the point of view of eliminating unnecessary complexity. Many readers may already be familiar with AD, but for those who are not, Section 3 gives a brief introduction to the details of the design axioms that are the core of this theory.

Many applications of AD in product design, system design, organisational decision making, and software development have appeared in the literature. AD was first applied in software engineering by Kim et al. [23] and was first applied in system design concepts by Suh [40]. Do and Park [14] also introduced new concepts by applying AD specifically to software design. Designing software based on AD creates "uncoupled or decoupled interrelationships and arrangements among 'modules', and is easy to change, modify, and extend" [45].

Harutunian et al. [16] used the first ('independence') axiom of AD to evaluate design decisions that provide an optimal software development project sequence. Suh and Do [45] combined the independence axiom of the AD theory and object-oriented programming to design large-scale software development systems. They were able to shorten the lead-time of software, improve reliability, reduce costs, and increase productivity.

Chen et al. [11] used the independence axiom to build a hierarchical knowledge base system. They constructed a simulation model and combined it with a decision support system to illustrate the effectiveness of the proposed knowledge base system. Huang [17] extended the AD principles and defined two master domains: design workspace and review workspace. They investigated the relations between the two domains based on the independence axiom. Huang and Jiang [18] used fuzzy set theory and expressed past experiences and insights as the membership functions of design parameters and evaluation criteria.

Lindkvist and Söderberk [27] used the independence axiom of AD and robust design to compare and evaluate assembly concept solutions. Chen et al. [10] used the independence axiom to facilitate both the integration of existing software and the modification of software since changes in one module did not affect other modules. Chen and Feng [9] used the independence axiom to test a computer-aided design model whether the proposed model satisfied the independence axiom or not. Yi and Park [47] developed software to analyse and construct the design process according to the independence axiom of the AD theory.

Togay et al. [46] proposed a component-oriented approach based on the AD theory. In the study, the V-Model proposed by Suh and Do [45] was extended since the AD process model did not address component-level architecture issues. Kulaka [25] provides a comprehensive overview of the literature on AD theory and principles.

Suh [43] divides "the treatment of complexity" into two distinct domains: treating the complexity in the "physical domain" and treating it in the "functional domain." In the first domain most engineers, physicists and mathematicians consider complexity as an "inherent characteristic of physical things, including algorithms, products, processes, and manufacturing systems". The "functional" approach is to treat complexity as a relative concept that evaluates how well we can satisfy "what we want to achieve" with "what is achievable" [43]. By considering a GSD project as an artefact it may be possible to apply AD theory to the project, and increase the probability of satisfying all project requirements (i.e. the project always performs what it needs to do).

The remainder of this paper is organised as follows. In Section 2 we introduce a reference model for GSD projects and Extended AD theory and use this theory to address the complexity of GSD planning and development projects in Section 3. After these reviews, in Section 4 we use an upper bound estimation of the complexity of the design matrix (by applying a complexity measure well known from information theory, Kolmogorov complexity (KC), and use this as a proxy measure of AD theory's Information Content metric). Using this proxy it is possible to measure the complexity of the design of an object, whereupon in this article the objects of interest are the software project planning project and the software product development project itself (i.e. we are *not* talking about the complexity of the software product). In Section 5 we present two hypothetical examples to compare both good and bad GSD planning projects and development projects from the complexity point of view. In Section 6 we discuss the separation of management functions from operations, and in Section 7 we present conclusions and future research directions.

2. A reference model for global software development

Prikladnicki et al. [34] proposed a reference model for GSD based on the results of real GSD case studies. Their proposed reference model includes the organisational and the project dimensions:

Organisational dimension (Planning): Prikladnicki et al. [34] state that planning is important to properly organise and manage distributed projects. They identified the initial planning as a formal and basic stage to decide if a project can be distributed, how to plan