International Journal of Scientific & Engineering Research, Volume 3, Issue 5, May-2012
ISSN 2229-5518

Methodologies to Amalgamate Artificial Intelligence and Software Engineering

Charu Khatwani

Abstract— Software Engineering (SE) and Artificial Intelligence (AI) are the two different disciplines that have developed without much indulgence with each other. The intersection between these two disciplines is currently rare but growing at an enormous rate. Merging the concepts of these two trades can take us to a whole new world of automated systems where human effort is minimized. These systems are Intelligent Systems because they not only develop efficient programs but also learn from past experiences.

This paper intends to study the techniques developed in AI from the standpoint of their application in SE. One such effort is to improve the incremental approaches in SE with the intelligence possessed by Artificial Systems. In particular, the AI systems tend to be built incrementally – an evolutionary paradigm is used. The essence of these incremental methodologies is captured in the notion of RUDE cycles and its modifications. Through these software systems must strive to construct programs which are understandable and this means that programs must be built "in the image of the human mind."

Index Terms— Knowledge-based Systems, Malleable Software, OSCON interface, Software Development Life Cycle, Waterfall Model.

1 Introduction

Artificial Intelligence is the area of computer science focusing on creating machines that can engage on behaviors that consider human intelligence. The dream of smart machines since decades has now become a reality through Artificial Intelligence. It aims to improve machine behavior in tackling complex tasks.

Software engineering is that form of engineering that applies the principles of computer science and mathematics to achieving cost-effective solutions to software problems. Thus, Software Engineering is about creating high-quality software in a systematic, controlled, and efficient manner.

The goal of this research paper is to bring the two ends together to make a robust, reliable and well defines skeleton of conventional software. To this framework we can attach the AI muscles that will provide greater power to the overall system but need to be closely confined and controlled if they are to be developed effectively.

2 Artificial Intelligence Software

"An evolutionary paradigm"

The AI Software provides us with the conventional software skeleton with the AI muscles making it reliable, robust and automated. This is based on the various paradigms and methodologies.

2.1 RUDE cycle

The AI systems tend to be built incrementally. The essence of these incremental methodologies is captured in the notion of RUDE cycle.The fundamental elements of all incremental system development procedures are:

- **R**un the current version of the system.
- **U**nderstand the behavior observed.
- **D**ebug the underlying algorithm to eliminate undesired be havioral characteristics and introduce missing, but desired, ones.
- **E**dit the program to introduce the modifications decided upon.

There are certain **questions** associated with the RUDE cycle.

2.1.1 How do we START?

We can construct a first version of the proposed AI software in much the same way as we construct prototype systems in conventional software engineering. We are aiming only for a software system that is an adequate approximation to the initial problem, and adequacy is primarily a behavioral characteristic.

- *Charu Khatwani is currently pursuing bachelors degree program in Computer Science and engineering from S.R.M.S.W.C.E.T under Uttar Pradesh Technical University, India, PH-919690687458.*
E-mail: charukhatwani@gmail.com

ISSN 2229-5518

Our AI-system prototype is thus not a prototype at all: it is a first version of what will subsequently become an adequate, practical AI system.

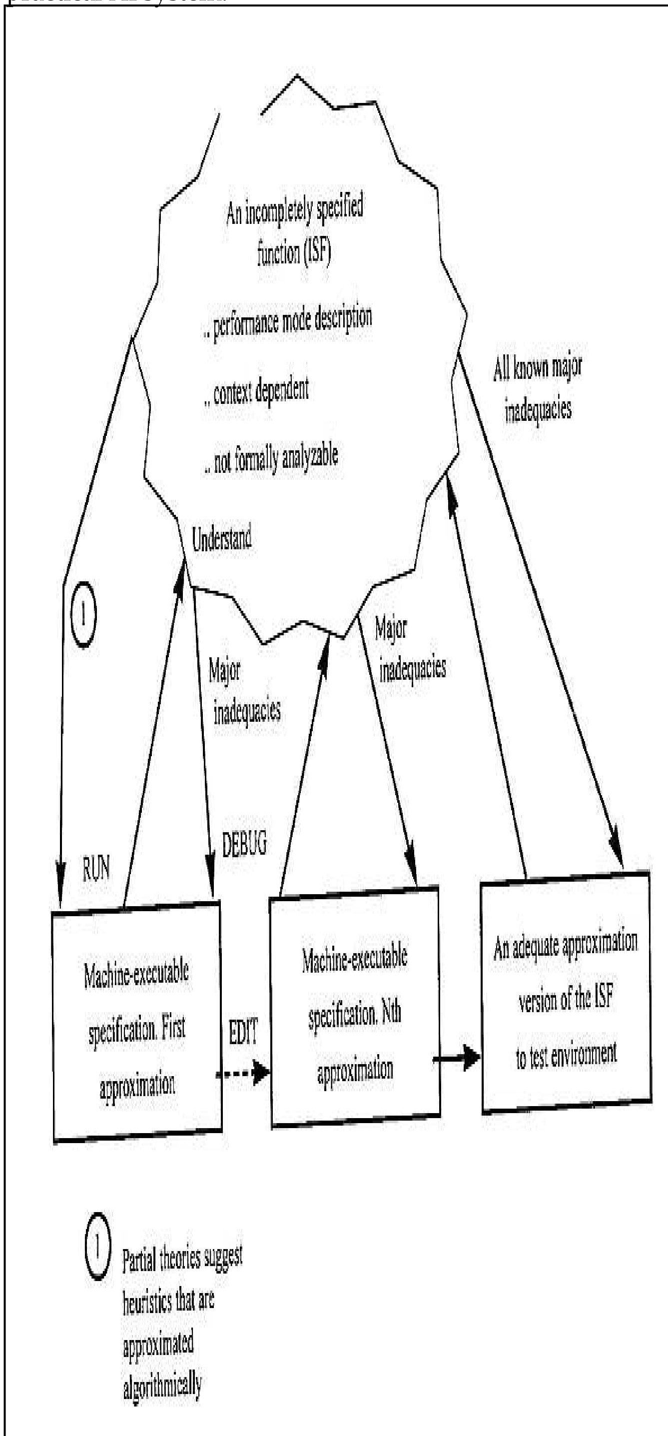

Fig. 1. The RUDE Cycle.

A major requirement of the first version system is that it should be readily **modifiable**. A second crucial requirement is that it does **run**.

Thus, we aim to form a Malleable Software. **Malleability**, as a characteristic of software systems, has two main aspects: we must be able to understand the working of the system and hence make good judgments about what changes to introduce; and the system must exhibit a maximum of functional de-coupling—so that the identified change can be introduced with minimal effect on other sub functions that we do not wish to modify.
These two aspects are both supported by a modular approach to system design.

2.1.2 How do we PROCEED?

Let's assume that we have our first version of software up and running on a computer system.
The task now is to understand the behavior and modify the current version to give a more appropriate subsequent version.
But the **Second Law of Program Evolution** states that—

"As a large program is continuously changed, its complexity, which reflects deteriorating structure, increases unless work is done to maintain or reduce it."

This is clearly depicted by the figure below—

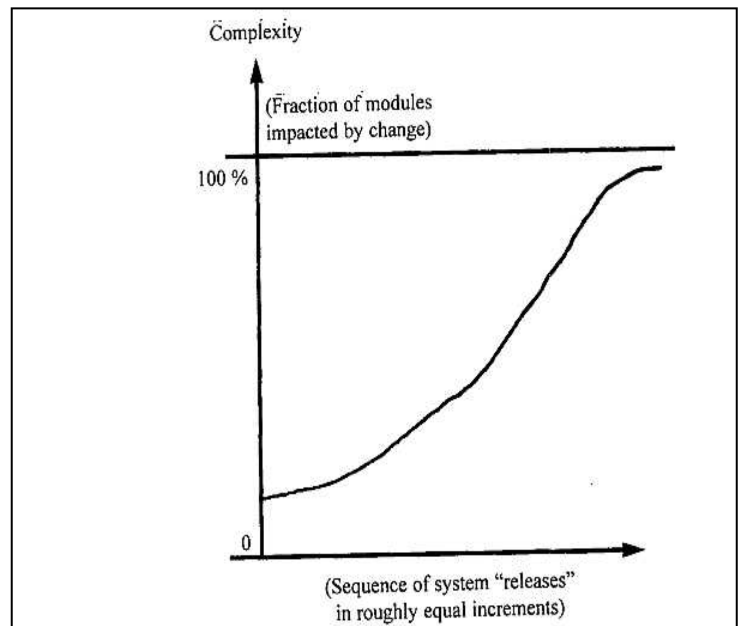

Fig. 2. Software System complexity as a function of successive modification from Lehman and Belady,1985, pg 387.

ISSN 2229-5518

A ***key to locking out the bad influence of the second law of program evolution*** is to introduce modifications at the highest level possible in the design structure of the system. When we have analyzed a problem and understood the fault in the current version, we do not dive into the code and modify it (unless the fault is a trivial code-level fault). What we should do is go back to the design documentation and determine where this fault was first introduced into the system—it may be a basic design fault, it may be a fault in the detailed design, or it may be simply a coding fault. When we have located the origin of the fault, we modify the system to *"**remove it at the source.**"*

Having corrected the problem at its origin we must then re-elaborate the changed design to determine the actual code changes that it specifies. So, the program code does, of course, become changed, but only as a result of a systematic redesign of the modified system.

2.1.3 How do we FINISH?

The finishline of the cycle is addressed in the guise of '**validating expert system**'. An expert system has to be validated (and sometimes verified—**V&V**) before it is fit any real use.

Verification confirms **"we have built the system right"**
Validation confirms **"we have built the right system"**

Validation is a stopping test.Successful validation is not necessarily the cue for a full stop on the versioning cycle, it is more the sanctioning of a move from experimental to real-world application.

2.2 WIZARD-OF-OZ

The next enhancement of the RUDE cycle is Wizard-of-Oz[4].It focuses on working of the prototype to the full as early as possible by making .The system stubs active rather than passive.

An **undeveloped AI muscle** (which supports the strong skeleton of Conventional Software) can be replaced by a 'hidden' human who supplies, on demand, appropriate, intelligent information. More mature readers will recall the hidden human who was in fact the 'brains' of the Wizard of Oz and hence the name of this approach. The point of the '**hidden**' Wizard is so that the behavior of the prototype can be under normal conditions—i.e, without system users reacting unnaturally to the knowledge that there is another human 'in the loop.'
This proposal has yet to be evaluated comprehensively, but has been employed with success in the development of the **OSCON interface to UNIX.**

The general strategy of isolating the AI muscles from the skeleton proves the **strong modular approach** of the AI System. First versions of the system can employ very basic, non-

heuristic rules in order to facilitate the construction of a strong skeleton within which heuristic rules and even heuristic reasoning mechanisms can later be introduced.

As there is no control to the number of iterative procedures in the RUDE cycle as well as Wizard-of-Oz so an improved methodology termed ad POLITE methodology was introduced.

2.3 THE POLITE METHODOLOGY

Bader, Edwards, Harris-Jones, and Hannaford [2]maintain that "knowledge-based systems" (KBS) are not yet associated with an accepted software development life cycle (SDLC) They make the point that the major system development feature that KBSs bring to the fore is a necessity for iterative refinement, and the RUDE cycle explained above is a likely basis for the required methodology.

The two fundamental flaws are singled out for treatment: "an engineering methodology for KBSs needs to provide a method for constructing an initial system from which the RUDE cycle can begin, as well as encompassing some form of control of the iterative process of development."

Rather than correcting the two flaws stated above the POLITE methodology merges the stuctured and controlled framework with the iterative RUDE cycle.

They select the waterfall model of software development as the "structured and controllable framework," and what was RUDE (plus waterfall) becomes POLITE, which is derived from:

Produce Objectives - Logical/Physical Design - Implement - Test - Edit

and, as we all know, POLITE is always preferable to RUDE. This is, of course, a slightly underhand way to introduce a possible enhancement. (But one that the RUDE paradigm seems to encourage—see, for a further example, *Mostow's [1985] Courteous* alternative, and, as a counterexample, *Trenouth's [1990b] VERY RUDE development.*)

This scheme is definitely waterfall based. The methodological enhancements come with elaboration of the infrastructure. To begin with: each step in the waterfall is now divided into two.

"**The left side is related to conventional components and the other side to the knowledge-based or cognitive elements of the system**"

For, as they state, the methodology as designed to deal "**with the development of hybrid systems comprising both conventional and heuristic components**".

ISSN 2229-5518

WATERFALL MODEL(STUCTURED AND CONTROLLED FRAMEWORK) + RUDE CYCLE

POLITE (PRODUCE OBJECTIVES - LOGICAL/PHYSICAL DESIGN - IMPLEMENT - TEST – EDIT)

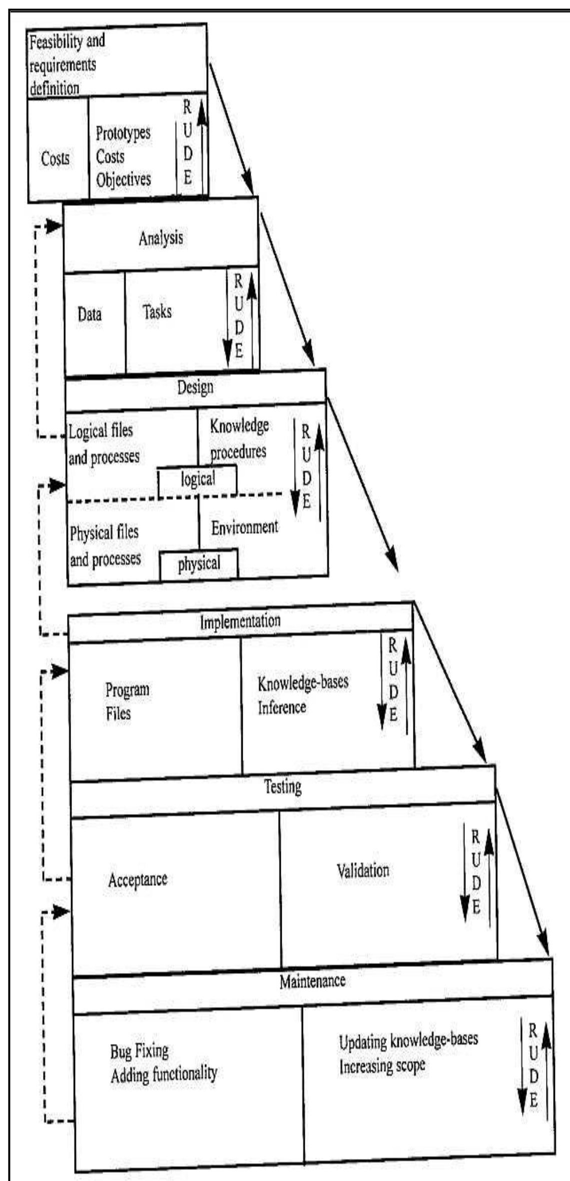

Fig. 3. The POLITE Life-Cycle [2]

3 CONCLUSION

This paper tends to study and combine the various methodologies used in Artificial Intelligence using the techniques and the different Software Development Life Cycle models in Software Engineering. The essence of these incremental methodologies is captured in the notion of RUDE cycles and its modifications. It is a first attempt to study the various methodologies of AI using SE and their improvements under one root.

ACKNOWLEDGMENT

I would like to appreciate Dr. Harshita Kumari for her help with this paper. Her encouragement and long hours with me in this effort are greatly appreciated. My thanks also go to Er. Rati Agarwal for her foresight and involvement in my research interest.

REFERENCES

[1] *Balzer, Robert, Thomas E. Cheatham, Jr., and Cordell Green, (1983) "Software Technology in the 1990's: Using a New Paradigm," IEEE Computer, November, pp. 39-45.*
[2] *Partridge, D., (1991) A new guide to artificial intelligence, Norwood, NJ: Ablex Publishing Co.*
[3] *Arango, G., Baxter, I. and Freeman, P. (in press) A framework for incremental progress in the application of artificial intelligence to software engineering, in D. Partridge (Ed.), Artificial Intelligence and Software Engineering, Norwood, NJ: Ablex Pub. Corp*
[4] *Analysing coherence of intention in natural language dialogue by paul mc kevitt, B.Sc. (Hons.) (Dublin), M.S. (New Mexico).*
[5] *Gilb, T. (1988) Principles of Software Engineering Management, Reading, Mass.: Addison-Wesley.*