Software Process Selection based upon Abstract Machines for Software Process Models

Shi-Kuo Chang, Jinpeng Zhou, Akhil Yendluri and Kadie Clancy
Department of Computer Science
University of Pittsburgh, Pittsburgh, PA 15238, USA
{schang, jiz150, aky13, kdc42 }@pitt.edu

***Abstract*— The Abstract Machine Model was developed by Chang to formalize the decision cycles of slow intelligence systems. It turns out the selection of software process can also be regarded as a slow intelligence system. In this paper we formalize abstract-machine prototypes for different software process models such as waterfall model, incremental model, spiral model, extreme programming model and scrum model. A Software Process Generator SPG was implemented to generate software process models based upon design considerations. Initial evaluation by undergraduate students using SPG to learn software processes suggests further improvements to make it a useful learning tool.**

Keywords—slow intelligence system, software process models, abstract machine model, component-based software engineering, software process learning tool.

1 Introduction

The slow intelligence system is an approach to design human-centric psycho-physical systems. A slow intelligence system (SIS) is a system that *(i) solves problems by trying different solutions, (ii) is context-aware to adapt to different situations and to propagate knowledge, and (iii) may not perform well in the short run but continuously learns to improve its performance over time*. The general characteristics of a slow intelligence system include enumeration, propagation, adaptation, elimination, concentration and multiple decision cycles [1]. In our previous work, an experimental test bed was implemented that allows designers to specify interacting components for slow intelligence systems [2].

In this paper, we formalize abstract-machine prototypes for different software process models such as waterfall model, incremental model, spiral model, extreme programming (XP) model and scrum model. Inspired by recent research on design spaces [3, 4, 5], a software process design space characterized by eleven parameters can be used to assist the designer in finding an appropriate software process model.

The paper is organized as follows. Section 2 presents an abstract machine model for the computation cycles. Section 3 shows some preliminary work based on building finite state machine (FSM) for each model. Based on the observations of preliminary work and the abstract machine of slow intelligence system (SIS), we further describe our abstract machines in Section 4. In Section 5 we present five prototypes to show how to use our abstract machine definition for different models. Once the abstract machine model is provided, a compiler can be constructed to generate the components. In Section 6 we describe the major steps of the generic Abstract Machine Compiler (AMC). Section 7 describes the Software Process Generator SPG we implemented to construct different process models based upon design parameters. Initial experimental results. discussion and conclusion are presented in Section 8.

DOI reference number: 10.18293/SEKE2018-135

2 The Abstract Machine Model for Computation Cycles

An SIS typically possesses at least two decision cycles. The first one, the *quick decision cycle*, provides an instantaneous response to environmental changes. The second one, the *slow decision cycle*, tries to follow the gradual changes in the environment and analyze the information acquired from the environments or peers or past experiences. The slow/quick decision cycles enable the SIS to both cope with the environment and meet long-term goals.

Complex SISs may possess multiple slow decision cycles and quick decision cycles. Most importantly, actions of slow decision cycle(s) may override actions of quick decision cycle(s), resulting in poorer performance in the short run but better performance in the long run.

To model such decision cycles we introduce an abstract machine model of multiple computation cycles.

The Abstract Machine Model is specified by: (P, S, P0, $Cycle^1$, ..., $Cycle^n$), where

P is the non-empty problem set,
S is the non-empty solution set, which is a subset of Po,
P0 is the initial problem set, which is a subset of P,
$Cycle^1$, ..., $Cycle^n$ are the computation cycles.

Each computation cycle will start from an initial problem set and apply different operators such as $+adap_{Aij}=$, $-enum<$, $>elim-$, $=prop_{Aij}+$ and $>conc=$ successively to generate new problem sets from old problem sets until a non-empty solution set is found. If a non-empty solution set is found, the cycle is completed and later the same computation cycle can be

repeated. If on the other hand no solution set is found, a different computation cycle is entered.
As an example the problem set P consists of problem elements p1, p2, p3, ..., pn, and each problem element pj is specified by a vector consisting of attributes Aij. A computation cycle x will attempt to find a solution set by first adapting based upon input from the environment: P^x0 ***+adap***$_{Aij}$***=*** P^x1 is to adapt based on attribute Aij, for example, by appending Aij to each element in P^x0 to form P^x1. Then it may try to find related problem elements: P^x1 ***-enum<*** P^x2 where P^x2 = {y: y is related to some x in P^x1, e.g. d(x,y) < D}.

Next it may try to eliminate the non-solution elements:
P^x2 ***>elim-*** P^x3 where P^x3 = {x: x is in P^x2 and x is in S}

Finally the solution elements (or alert messages if there are nosolutions) may be propagated to peers: P^x3 ***=prop***$_{Aij}$***+*** P^x4 is to export/propagate attribute Aij to peers.

Therefore this computation cycle can be specified succinctly as follows: Cycle^x [guard x,y]: P^x0 ***+adap***$_{Aij}$***=*** P^x1 ***-enum<*** P^x2 ***>elim-*** P^x3 ***=prop***$_{Aij}$***+*** P^x4.

The above expression is a specification of the computation cycle, not a mathematical equation. This expression should be read and interpreted from left to right.

If P^x4 is non-empty, the Abstract Machine will complete this cycle of computation and terminate at the end of Cycle^x, and it may later resume at the beginning of Cycle^x. Otherwise P^x4 is empty and the Abstract Machine will jump to a different Cycle^y. This is specified by [guard x,y] where x is the current computation cycle if a solution set is found (P^x4 is non-empty), and y is the computation cycle to enter if no solution set is found (P^x4 is empty). Before an Abstract Machine completes its current computation cycle, it will propagate the solution set (or alert messages) to its peers.

In the above, the elimination operator can be replaced by the concentration operator, whenever the solution set is not known apriori. The concentration operator applies a predefined threshold to filter out problem elements below the threshold:
P^x1 ***>conc=*** P^x2 where P^x2 = {x: x is in P^x1 and th(x) above a predefined threshold t}.
.

3 Software Process Models

Software process models provide certain workflows for software development. Intuitively, we can use finite state machine (FSM) to illustrate these models. Each step in the software process can be represented as a state in FSM. The inputs and outputs of each state are corresponding to the requirements and products of each process step, which may include documents, program codes, user communications, test datasets, prototypes, and timings.

3.1 FSM for different models

Based on the state transition tables, we drew the sketches for five software process models. These are meant to illustrate the concept, and the specific details of each software process model may vary.

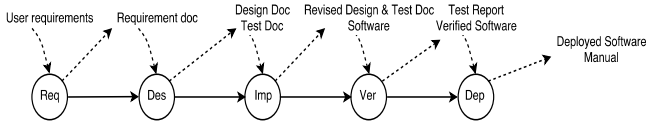

Fig 1. Waterfall model

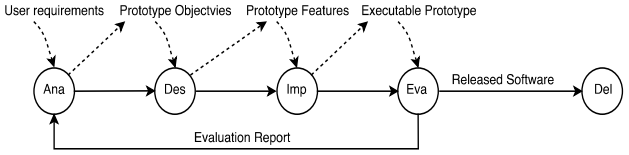

Fig 2. Incremental model

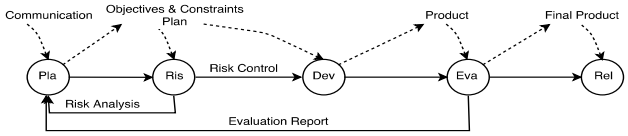

Fig 3. Spiral model

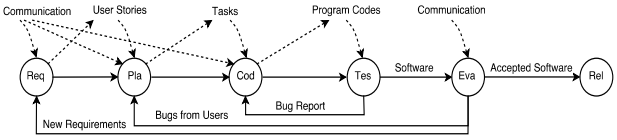

Fig 4. Extreme programming model

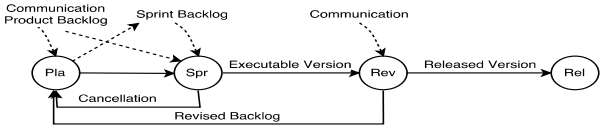

Fig 5. Scrum model

3.2 Observations

There are some key observations that inspire the definition of our abstract machines:

- All process models are based on a major workflow, starting from the user requirements to the final release of software. Thus, we can use operation cycles to represent the process flows. Furthermore, we need to bind to a starting point so that the machine starts from the first user requirements.

- The final purpose of a software process is to build a production software, which typically consists of different features, or objectives. These objectives derive from the original user requirements and are abstracted, implemented, and verified during the process. Thus, each state in the FSM actually can be presented as a set of

objectives, which is similar as the problems set in SIS abstract machine.

- An objective has a lifetime starting from user requirements to verification. Each step in the process will update or mark the objective with a new state. Thus, we can assign a color for each objective to represent its states during the entire process.

- The software engineering or project managing operations during the process can be represent by the operators. These operations can perform add/delete/update on each objective, including coloring.

- At some points, a step may have different successors based on certain situation. Thus, we also need a guard function to provide process control. Furthermore, we need to know these specific steps and situations.

- Different Models may have different behaviors in similar step. e.g., agile models do not need explicit and complete user requirements and system design.

4 Abstract Machines for Software Process Models

4.1 Machine definition

Based on the observations in Section 3.2, we can define the abstract machine, by modifying the abstract machine for SIS:

$$M_{spm} = \{P, S, P_0, Cycles\}, starting\ from\ cycle_1$$

Where P is the objective set, P_0 is the initial set. S is the solution set which includes all objectives that have been implemented and verified. $Cycles$ are sequences of different operators. As mentioned above, this abstract machine should start from a certain starting point. By default, we set the starting point to be the beginning of cycle_1.

4.2 Objectives

An objective is corresponding to a certain user requirement or feature for the target software. We define four colors for each objective.

- White: it's abstracted or included into the software process.
- Yellow: it's implemented and ready for verification.
- Red: failed in verification, such as failed in testing or user acceptance.
- Green: it's verified and ready to be used.

4.3 Guard function

In order to support process control, we also define a guard function by extending the SIS abstract machine's guard function:

guard [a, b, P_checkpoint, constraint, P_newInit]

Where *a* is the current cycle, *b* is the target cycle. When it reaches the *P_checkpoint* of cycle *a*, it will check whether *a* certain constraint is satisfied. If so, it will jump to start cycle *b* with *P_newInit*.

4.4 Operators

To provide a general definition, we defined six basic operators for software process models:

- Abstract (**abst**): Translate user-described requirements to software-engineering requirements. This operator will initialize objectives with color white. We further divide this operator into two types: (1) *Enumerative Abstract* (-**abst**<). This type will process every objective; (2) *Selective Abstract* (=**abst**=). This type will process selected objectives only. It does not guarantee that all objectives will be processed.

- Design (**desi**): Functionalize non-green objectives to module-level or function-level objectives. It also has two types: enumerative (-**desi**<) and selective (=**desi**=).

- Implement (=**impl**=): Implement white/red objectives to real product-level objectives and color them as yellow. We assume that implementation is strictly bound to the objectives. E.g., each objective will be implemented as a module. Thus, only selective is required here.

- Test (=**test**=): Validate yellow objectives and color them as green or red. Similarly, only selective is required.

- Adjust (=**adju**=): Modify objectives based on (external) non-engineering issues, such as user communications, funding issues. This operator is essential for agile models.

- Release (=**rele**+): release all green objectives. This operator is similar as the propagator in SIS abstract machine, which generates some outputs to environment.

5 Abstract Machine Prototypes

Based upon the observations in Section 3.2 and the formal specification of abstract machines in Section 4, the software process models can now be formally specified. Again, these are meant to illustrate the concept, and the specific details of each software process model may vary.

5.1 Waterfall

Prototype:

- Cycle_1: guard[1, 2, P2, NULL, P2]
  - P0 **-abst<** P1 **-desi<** P2
- Cycle_2: guard[2, 2, P2, has_non-green, P2]
  - P0 **=impl=** P1 **=test=** P2 **=rele+** P3

In cycle_1: it requires a complete abstraction and design.

In cycle_2: it will go through implementation, test, and final release. Whenever there's a failed objective after test, it should go back to the implementation and redo the following process again.

The machine halts at P3 in cycle_2.

5.2 Incremental

Prototype:
- Cycle_1: guard[1, 2, P2, NULL, P2]
  - P0 **=abst=** P1 **-desi<** P2
- Cycle_2: guard[2, 2, P2, one_non-green, P2], guard[2, 1, P3, all_green, NULL]
  - P0 **=impl=** P1 **=test=** P2 **=rele+** P3

In cycle_1: the abstraction can be incomplete. But the design should take care of all current objectives.

In cycle_2: different from waterfall model, here it will go back to cycle_1 for next increment when the current increment is finished.

The machine halts when no more increment is required, which means P0 in cyle_1 is empty.

5.3 Spiral

Prototype:
- Cycle_1: guard[1, 1, P3, one_red, NULL], guard[1, 2, P3, no_red, P1]
  - P0 **=abst=** P1 **=impl=** P2 **>+adju=** P3
- Cycle_2: guard[2, 1, P5, all_green, NULL]
  - P0 **-abst<** P1 **-desi<** P2 **=impl=** P3 **=test=** P4 **=rele+** P5

In cycle_1: it required to build a simple prototype to evaluate the risk. Thus, we need an implement and adjust operator here. If the risk evaluation says good, then it will transfer to the cycle_2 for further process.

In cycle_2: similarly, it will go back to cycle_1 until there's no more work to do.

The machine halts when P0 in cyle_1 is empty.

5.4 Extreme programming

Prototype:
- Cycle_1: guard[1, 2, P2, NULL, P2]
  - P0 **=abst=** P1 **=desi=** P2
- Cycle_2: guard[2, 2, P2, **hours**, P2], guard[2, 3, P3, **days**, P3]
  - P0 **>+adju=** P1 **=impl=** P2 **=test=** P3
- Cycle_3: guard[3, 1, P2, non-empty, P2]
  - P0 **=rele+** P1 **>+adju=** P2

In cycle_1: it does not require complete requirements or design. It selects a certain user story to implement.

In cycle_2: The process is controlled based on timing. Thus, the major constraint here should be related to the specific time deadline. Furthermore, we need the adjust operator to make sure the implementation and testing are sensitive to user communications.

In cycle_3: after few days, it's supposed to generate a version for quick release. Then it can keep finishing rest or new objectives based on feedbacks.

The machine halts when P2 in cycle_3 is empty, which means feedbacks confirm that no more objectives.

5.5 Scrum

Prototype:
- Cycle_1: guard[1, 2, P2, NULL, P2]
  - P0 **=abst=** P1 **=desi=** P2
- Cycle_2: guard[2, 2, P2, **day**, P2], guard[2, 3, P2, **Weeks**, P2]
  - P0 **=impl=** P1 **=test=** P2
- Cycle_3: guard[3, 1, P3, non-empty, P3]
  - P0 **>+adju=** P1 **=rele+** P2 **>+adju=** P3

In cycle_1: it does not require complete requirements or design. It selects a certain backlog and launch it as a sprint.

In cycle_2: it starts a sprint. Inside this sprint, no requirement modification is allowed.

In cycle_3: in the end of a sprint, the developing team will review the sprint. Then, based on user communications, the set of objectives (backlogs) will be updated.

The machine halts when P3 in cycle_3 is empty, which means all backlogs are finished.

6 The Software Process Model Generator

The Abstract Machine Model is a formal specification of the computation cycles of a slow intelligence system. Once the abstract machine model is provided, a Software Process Generator SPG can be used to generate software process model based upon design considerations.

The input to the SPG are the various software process models SPM specified by Abstract Machines. The user/designer can interact with SPG to select the appropriate design choices. After a software process model is selected, the output in the form of a web page is generated by SPG. This web page describes the software process and can be used by the user/designer to track a project according to the selected process model.

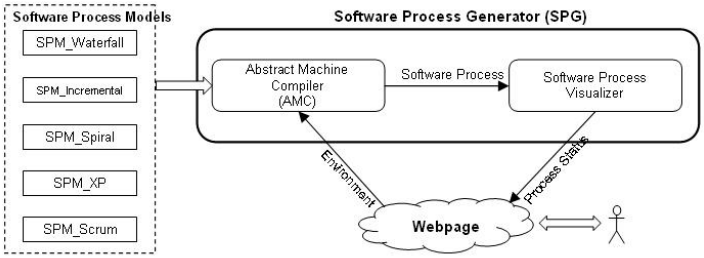

Figure 6. The software process model generator.

User/designer interacts with SPG through the webpage, such as tracking process status, updating environment and so on. The user/designer can make design choices, update requirements if possible, modify environment, etc. The SPG updates the current software process status based on the the selected SPM, and user interactions.

In what follows we describe the major steps of how the AMC and visualizer work in SPG.

Step 1: Adapt inputs from the user
The AMC will first invoke an interface to receive a set of features from the user/designer. This set may include necessary information of the project for simulating different process models. It is submitted by the user/designer through a webpage (see box below).

In practice, the user/designer makes choices to specify the desired parameters (see Figure 7 in Section 7).

```
Interactor:
        // adapt current data with user inputs
        while(user inputs are not required) {
                sleep();
        }
        while(currentData is not updated) {
                adaptCurrentDataWithInputs();
`       }
        send DATA_READY msg to Controller;
```

AMC Controller manages the process by controlling the state machine (see box below).

```
Controller:
        // manage the process of one process model
        while(true) {
                while (stateMachine.precheck()) {
                        trigger Interface,
                        // wait for user inputs
                        stateMachine.wait(DATA_READY);
                }
                stateMachine.perform(currentData);
                send msg to visualizer if necessary;
        }
```

The State Machine will determine the action and the output. It may give several tries. For example, two solutions can be applied to one certain state when given certain input (see box below).

```
State Machine:
        // manage the states
        enum Status {
                State0,
                State1,
                …;
        }

        State currentState = State0;//initial state

        bool precheck(); // return true if current state requires inputs

        int wait(msg); //wait until certain msg is received

        void perform(Data currentData) {
                // based on certain state and certain input
                switch (currentState) {
                        case 'State0':
                                perform accordingly;
                                break;
                        case 'State1':
                                perform accordingly;
                                break;
                        ….
                }
        }
```

Step 2: Simulate process models
The AMC is responsible to simulate every model.

To answer the question "Which process model is the best?": (1) each state will be measured to make sure the current project status is in a "safe-zone". Depending on the results of the measurement, either enumeration operator or elimination operator can be applied; (2) a specific function, which takes certain parameters into consideration, will be applied to evaluate the performance of each model.

If a certain model is simulated successfully, the evaluation results and simulation logs will be presented to the user on demand. If a certain state of model A violates project's configuration, the AMC will terminate A's simulation and start the next un-simulated model.

Step 3: Model selection
After all models have been simulated, the AMC will choose the model with the best evaluation result to the user, and present it as the optimal solution to the user.

Step 4: Model visualization
A process visualizer is built to show the simulation to the user. There are three cases that the visualizer is invoked: (1) the running simulation requires dynamic or runtime inputs from users; (2) the user requests to check current simulation status; (3) the AMC has decided the optimal solution (see box below).

```
Visualizer:
        // present the AMC results
        void showCurrent(); // invoked by the AMC or the user

        void showOptimal(); // invoked by the AMC
```

7 An Experimental SPG System

An experimental SPG was implemented. Software processes are characterized by the following eleven design parameters:

- Early Functionality (iteratively introduce features, only produce final product),
- Feature Adaptation (impossible, flexible),
- User Involvement (C only initially, at requests, frequent feedback),
- Documentation (not produced, produced),
- Experienced Team (requested, not required),
- Model Type (C linear, iterative),
- Planning and Scheduling (upfront, continuous),
- Risk Mitigation (yes, no),
- Project Size (C small, medium, large),
- Prototypes (used, not used).
- Cross-platform development (no, yes)

Parameter	Value
Early Functionality	◉ Iteratively introduce features ○ Only produce final product
Feature Adaptation	○ Impossible ◉ Flexible
User Involvement	(Initially) ○ 0% ○ 10% ○ 20% ○ 30% ○ 40% ○ 50% ○ 60% ○ 70% ○ 80% ○ 90% ○ 100% (Frequent feedback)
Documentation	◉ Not produced ○ Produced
Experienced Team	○ Requested ○ Not Required
Model Type	(Linear) ○ 0% ○ 10% ○ 20% ○ 30% ○ 40% ○ 50% ○ 60% ○ 70% ○ 80% ○ 90% ○ 100% (Iterative)
Planning and Scheduling	○ Upfront ○ Continuous
Risk Mitigation	○ Yes ○ No
Project Size	(Small) ○ 0% ○ 10% ○ 20% ○ 30% ○ 40% ○ 50% ○ 60% ○ 70% ○ 80% ○ 90% ○ 100% (Large)
Prototype	○ Used ○ Not Used
CrossPlatform	○ No ○ Yes

SubmitParameters

Figure 7. The designer specifies the needed parameters.

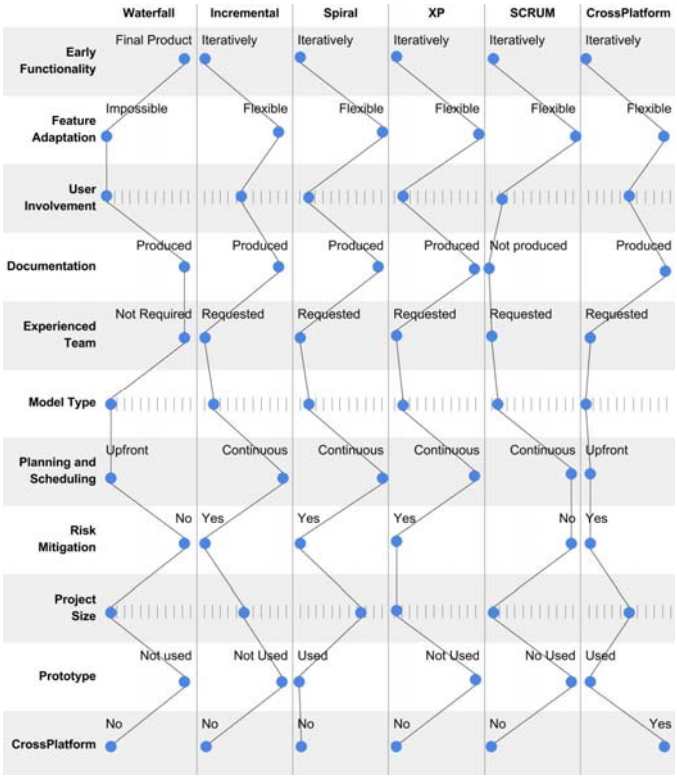

Figure 8. Stored profiles of software processes.

A parameter with continuous value is indicated by the letter 'C'. In practice the designer can specify a continuous parameter from 0% to 100% in 10% increments (see Figure 7). Parameters not specified by the designer are not used in calculating the optimum. The designer-specified profile (Figure 7) is compared with the pre-specified profiles of software process models (Figure 8) and the one with minimum distance is the SPG-recommended software process model (Figure 9).

waterfall : 3

incremental : 1

spiral : 1

xp : 1

scrum : 0 (optimal)

crossplatform : 1

Figure 9. Scrum software process recommended by SPG.

When the designer clicks on the link for Scrum, a Scrum software process simulator is provided. As shown in Figure 10 the designer can simulate the execution of the software process by clicking on the actions associated with its current state. In addition a tutorial on Zoho is provided as the recommended tool for Scrum software development.

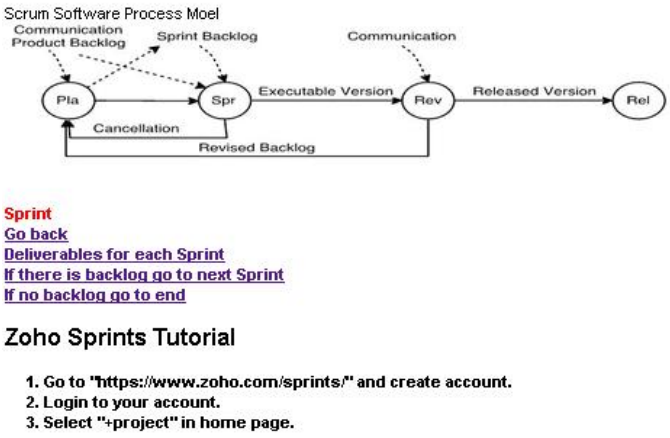

Figure 10. Scrum software process simulator.

8 Discussion

The Abstract Machine Model is a formal specification of the software processes, based upon which the SPG tool was implemented: http://ksiresearchorg.ipage.com/spg/. The SPG tool was used by 32 undergraduate student project groups in two software engineering classes at the Univ. of Pittsburgh to experiment with software processes. The students were then asked to evaluate the SPG. In response to the question whether the SPG tool enhanced understanding of the software processes, the average rating is 0.35, between “a lot” (0.5) and “a little” (0.25). There are comments that the individual model pages are the most helpful, and percentage as a parameter value is a little vague.

The current SPG was implemented with pre-defined web pages representing the states of different software process models. We are implementing a better version by dynamically generating customized web pages (the process states). Both pre-defined software processes as well as hybrid software processes can then be generated, thus making the SPG a more powerful learning tool. More information is added to the individual model pages, and parameters are better explained. With more improvements the SPG tool can become a valuable learning tool.

Acknowledgement:

This research was supported in part by KSI Research, USA.

References:

[1] Shi-Kuo Chang, "A General Framework for Slow Intelligence Systems", *International Journal of Software Engineering and Knowledge Engineering*, Volume 20, Number 1, February 2010, 1-16.

[2] Shi-Kuo Chang, Sen-Hua Chang, Jun-Hui Chen, Xiao-Yu Ge, Nikolaos Katsipoulakis, Daniel Petrov and Anatoli Shein, "A Slow Intelligence System Test Bed Enhanced with Super-Components", Proceedings of 2015 International Conference on Software Engineering and Knowledge Engineering, Pittsburgh, USA, July 6-8, 2015, 51-63.

[3] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan, The design of bug fixes, in Proc. Int. Conf. Softw. Eng., 2013, pp. 332341.

[4] Murphy-Hill, E., Zimmermann, T., Bird, C., and Nagappan, N., 2015. The Design Space of Bug Fixes and How Developers Navigate It. IEEE Transactions on Software Engineering 41, 1, 65-81.

[5] S. CARD, J. MACKINLAY, AND G. ROBERTSON. A MORPHOLOGICAL ANALYSIS OF THE DESIGN SPACE OF INPUT DEVICES. ACM TRANSACTIONS ON INFORMATION SYSTEMS, 9:99122, 1991.