**CS 2310 Project**

**Abstract Machine for Software Process Models**

Jinpeng Zhou / jiz150

Project advisor: Prof. ShiKuo Chang

Table of Contents

[1. Introduction 2](#_Toc500954821)

[2. Preliminary work 2](#_Toc500954822)

[2.1. FSM for different models 2](#_Toc500954823)

[2.2. Observations 3](#_Toc500954824)

[3. Abstract Machine for Software Process Models 4](#_Toc500954825)

[3.1. Machine definition 4](#_Toc500954826)

[3.2. Objectives 4](#_Toc500954827)

[3.3. Guard function 4](#_Toc500954828)

[3.4. Operators 4](#_Toc500954829)

[4. Abstract Machine Prototypes 5](#_Toc500954830)

[4.1. Waterfall 5](#_Toc500954831)

[4.2. Incremental 5](#_Toc500954832)

[4.3. Spiral 6](#_Toc500954833)

[4.4. Extreme programming 6](#_Toc500954834)

[4.5. Scrum 7](#_Toc500954835)

# 1. Introduction

In this project, we formulized abstract-machine prototypes for different software process models, such as waterfall model, incremental model, spiral model, extreme programming (XP) model, and scrum model.

Section 2 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 3. In Section 4, we presented five prototypes to show how to use our abstract machine definition for different models.

# 2. Preliminary work

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.

## 2.1. FSM for different models

Based on the state transition tables, we drew the sketches for five software process models.



Fig 1. Waterfall model



Fig 2. Incremental model



Fig 3. Spiral model



Fig 4. Extreme programming model



Fig 5. Scrum model

## 2.2. Observations

There are some key observations that inspires 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 start 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 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.

# 3. Abstract Machine for Software Process Models

## 3.1. Machine definition

Based on the observations in Section 2.2, we can define the abstract machine, by modifying the abstract machine for SIS:

Where is the objective set, is the initial set. is the solution set which includes all objectives that have been implemented and verified. are sequences of different operators. As mentioned above, this abstract machine should start at a certain point. By default, we set the start point to be cycle\_1.

## 3.2. 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.

## 3.3. 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.

## 3.4. 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:
  + Enumerative Abstract (**-abst<**). This type will process every objective.
  + 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.

# 4. Abstract Machine Prototypes

## 4.1. 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.

## 4.2. 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.

## 4.3. 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.

## 4.4. 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.

## 4.5. 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.