Formal Transformation of Requirements in Software Development Life Cycle

Abstract: Traditional Software Development Life Cycle (SDLC) is based on informal specification of requirements which are often ambiguous or inconsistent and informal development methods. This ambiguity leads to unverifiable requirements especially when developing critical systems. In this paper, we propose applying Formal Methods (FMs) to SDLC including requirements’ analysis, specifications derivation, and system designing, and validation. The proposed SDLC approach is referred to by Formal Transformed software Development Life Cycle (FTSDLC). Since FMs based on mathematics and logic, we expect our model to work successfully in developing critical systems.

Keywords: Software Development Life Cycle, Formal Methods, Critical Systems.

1. INTRODUCTION

Software Development Life Cycle (SDLC), relates to model or methodologies that people use to develop systems, generally computer systems [14]. SDLC is based on informal specifications which are often ambiguous or inconsistent [7]. Practically, SDLC begins with a customer statement of requirements, or CSR. Those requirements are expanded and amplified to give system specifications [4]. As the customer statement requirement is normally expressed in a natural language, such as English, it often ambiguous or involve contradicting requirements. In translating a CSR to a system specification, which expressed also in an informal notation, there is no guarantee that such ambiguities and contradictions will be recognize and thus resolved. The failure of many software-controlled systems may only cause inconvenience. In other words, it may cause no serious and long-term damage. However, there are some systems where failure can result in significant economic losses, physical damages or threats to human life. These systems are usually called critical systems. It is natural, then, that we should look to mathematical techniques to address such problems in software development.

The use of Formal Methods (FMs) offers the possibility to specify and develop systems, especially those systems of a high level of complexity [3]. In general, the term FM refers to the use of techniques that employ formal logic and desecrates mathematics in specification, design, and implementation of software (and hardware) systems [3]. This paper aims to highlight the value of applying FMs in the conventional software development life cycle techniques to make them more appropriate for developing a critical system.

2. FORMAL METHODS AND CRITICAL SYSTEMS

Formal methods allow a software engineer to create a specification that is more complete, consistent, and using conventional or object-oriented methods [3, 8]. Sets theory and logical notation are used to create a clear statement of facts (requirements in SDLC context). This mathematical specification can then be analyzed to prove correctness and consistency of those facts or requirements. Because the specification is created using mathematical notations, it is inherently less ambiguous than informal modes of representation [8]. There is a continuing debate in the critical systems community about the role of FMs in the safety-critical software development process. Nevertheless, in some cases, using FM is mandatory and a requirement. For example, the use of formal specification and associated verification is mandate in UK defence standard for critical software [1].

FMs may be used at two levels in the development of critical systems: (1) a formal specification of the system may be developed and mathematically analyzed for inconsistency, our work will be concerned in this area. (2) A formal verification that the code of a software system is consistent with the specification may be developed.

In [2], the authors report results of an investigation into the effectiveness of formal methods as an aid to the requirement analysis of critical systems, system-level fault-protection software on a spacecraft. A generic approach to the formal specification of system requirement has been presented in [6] and it has been proven that the model has a potential of reducing the effort to formally specify system requirement [6]. In [11], the author summarizes the challenges that faces
software development for critical systems and discuss some of the opportunities for meeting those challenges.

Formal methods have been successfully applied to practical safety-critical systems. For example, research groups at NASA and Rockwell Aviation have used formal methods to detect serious errors in requirements specifications of both the International Space Station (NASA) and a flight guidance system (Rockwell) [3]. Further, the Naval Research Laboratory has used formal methods to expose significant errors in a moderate-size contractor specification of a safety-critical weapons system [6].

3. TRADITIONAL SOFTWARE DEVELOPMENT LIFE CYCLE

Computer systems have become more complex and usually link multiple traditional systems often supplied by different software vendors (especially with the advent of Service-Oriented Architecture). To manage this task, a number of (SDLC) models have been created: waterfall, fountain, and spiral build and fix, rapid prototyping, incremental, and synchronize and stabilize. Although in the academic sense, SDLC can be used to refer to various models, SDLC is typically used to refer to a waterfall methodology [8]. SDLC describe phases of the software cycle and the order in which those phases are executed.

Besides the waterfall model, there are many models. In fact, many companies adopt their own models. However, all these models actually have similar pattern. The general, basic model is shown Figure 1. Figure 1 depicts the phases of the software cycle and the order in which those phases are executed. There are many models as we stated earlier. However, all available models follow and have similar pattern. Each phase of the four phases shown in figure 1 produces deliverables required by the next phase in the life cycle. Requirements are translated in to design. Code is produced during implementation that is driven by the design. Testing verifies the deliverable of the implementation phase against requirement.

As shown in Figure 2 our proposed FTSDL is based on incorporating formal logic and discrete mathematics in requirement, specification, design, and validation of software systems. FTSDL consists of the following phases (1) Formal requirement specification (2) Formal specification (3) Formal designing specification (4) Formal validation, and they are to be further explained in the following sections.

4. FORMAL TRANSFORMED SOFTWARE DEVELOPMENT LIFE CYCLE

Our Formal Transformed Software Development Life Cycle (FTSDL), shown in Figure 2, is an approach to develop critical systems which has something in common with the general SDLC shown in figure 1. The basic difference is that the development process is based on formal mathematical transformation in our proposal. Our proposal can be typically employed when developing system that requires strict safety, reliability, and security requirements.
4.2 Formal Specification

Formal specification uses mathematical notation to describe, in a precise way, the properties which an information system must have. This description is expressed without unduly constraining the way in which these properties are achieved [12]. In other words, formal specification tries to describe what the system must do without saying how to be done. To apply FMs efficiently in the specification phase one can go through the following steps:

(i) Writing a safety case which documents the evidences and arguments that the system is safe to operate. This step must be accomplished with care as logical fallacies in underlying argument might undermine the system’s safety claims. The elements of safety case document are shown in Figure 4.

(ii) Write the formal specification for the system. For example, the system specifications could be written in a formal notation such as Z. One particular benefit of using formal notation is that it forces us to consider all the cases that bring up most of the industrial error behaviours. This is an aspect often omitted from informal description.

(iii) Apply the safety case element to the formal specifications document to ensure that the desired proprieties of formal specification. Those desired properties are: consistency, completeness, and having no ambiguity. Consequently, claims in the safety case are in much higher likelihood of achieving. See Fig. 5.

4.3 Formal Design Specification

In requirements engineering process, there are three levels of software specifications which must be developed [11]. These are: user requirement, system requirement and software design specification. To design a system formally we need to consider the following:

(i) The first step of formal designing should be concerned with decomposing the system into a set of sub-systems that are to be developed independently, see Figure 6.

(ii) The sub-systems themselves have to be decomposed and organized into an architectural design and an interface design as well.

(iii) Concurrently, (see Figure 6):

a) Applying model checking step is required to build a finite model of the system to ensure that the desired properties hold in the model. Model checking is important in the checking of hardware design if the state space is sufficiently small to make this feasible [8].

b) Applying theorem proving is also required to present both the system and its desired properties as formulas in the form of mathematical logic. However technique such as a Binary Decision Diagrams (BDDs) allows impressively large number of state to be handled [8]. This makes this step infeasible unless the system being developed is mission-critical.

c) Equally important, the following tasks are needed for each interface:

1. Determining the type of interface. The interface in this section is categorized in Table 1, according to their role in the system.

2. Providing interface specification for each interface to enable developers to know what services will be available and how these services
can be accessed. An interface specification could be produced by defining a set of abstract data types or objects. See Figure 7.

3. Apply algebraic approach for each interface to define the abstract data type. In an abstract data type, the type is defined by specifying the type operations rather than the type presentations. Since LARCH is probably the best known language for algebraic specification, it can be used for interface specification.

4. Combine the specification for each interface type in the same sub-system, and then combine the whole subsystems specifications. See Figure 6.

**Table 1. Interface types**

<table>
<thead>
<tr>
<th>Interface Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. User Interface</td>
</tr>
<tr>
<td>2. Hardware Interface</td>
</tr>
<tr>
<td>3. Software Interface</td>
</tr>
<tr>
<td>4. Communication Interface</td>
</tr>
</tbody>
</table>

**Figure 6. Formal design specification design process**

**Figure 7. Interface Objects**

### 4.3 Formal Validation

Any specification needs to be validated against the customer requirements. This is generally done via some form of review or walk-through [4]. For this it is required during this phase to ensure that the mathematical formula, equations etc. capture the intention of the requirements. The advantage of this validation is that any question about what will happen for certain input can be precisely answered by carrying out some mathematics. Formal specification can, thus, be proved to be consistent. It is recommended to use rapid prototyping to validate the specification. This prototyping is based on executing the specification which depends upon the used specification type.

The following methods could be used to execute specification:

- Method (1): Specification expressed using axioms can be executed using rewrite interpreter [9].
- Method (2): for specification expressed in logic, some kinds of theorem prover must be used. One can view logic programming languages such as Prolog, as a way of executing formal specification.

### 5. DISCUSSIONS

We have presented a transformation approach to formal specification of software development life cycle. The proposed approach starts with a formal requirement, see figure 3, to obtain a standard written document and a graphical modal document. The proposed approach also involves a Requirement Validation examination to ensure that all system requirements are stated unambiguously. The approach helps formally specifying the properties of a system during the specification phase.

In developing a given system, the system can be decomposed into a set of sub-systems and formal designing techniques can be applied to design and verify each sub-system using modal checking and theorem proving. Formal validation can be applied to prove system consistency.

Next we list the following observations that can be made:

Observation 1: Using FMs for specifying and developing software system is not simple. However, FM promises producing more complete and consistent system compared to the systems developed by using conventional or object-oriented methods.

Observation 2: It should be recognized that Formal methods can equally perform well when applied to hardware as well as software development.

Observation 3: There are occasions however where formal methods are not only desirable, but even positively required. Thus, we strongly we recommended the use of formal methods in certain classes of applications, especially when developing critical systems.

Observation 4: As stated earlier, the use of formal methods works successfully developing critical systems because it is based on logic and mathematics. This mathematical specification can then be analyzed to prove correctness and consistency.

Observation 5: The expertise required for the mathematical notation used for the formal transformation adds to system development complexity and, thus, increases the cost of developing software and hardware
systems. This observation makes this model impractical for the development of simple and non-critical systems.

6. CONCLUSIONS

In this paper, the Formal Transferred Software Development Life Cycle (FTSDLCL) model is proposed. It has been showed that Formal methods do provide a mean by which critical systems can be formally developed so there correctness is achieved. We believe that applying the Traditional Software Development Life Cycle with the use of Formal Methods promises producing better results when applied for developing critical systems. Further, Formal Methods can be equally applied to hardware design as well as to software development.

7. REFERENCES


