On the Use of Model Checking in Verification of Evolving Agile Software Frameworks: An Exploratory Case Study

Nan Niu, Steve Easterbrook

Department of Computer Science, University of Toronto
Toronto, ON, Canada M5S 3G4

Abstract. Evolution is a basic fact of software life. Domain-specific agile software frameworks are key to modern enterprise information systems (EIS). We propose a model checking approach to formal verification of agile frameworks that evolve continuously. The results obtained can be used to justify the maintenance activities in software evolution and identify important but implicit assumptions about the application domain of the framework. An industrially relevant exploratory case study is conducted to validate our hypothesis and proactively direct future research.

1 Introduction

Agile software frameworks capture the commonalities in design and implementation among a family of related applications. Software engineers use frameworks to reduce the cost of building complex systems. Frameworks promote reuse and rapid development by constraining the space of possible solutions. Formal methods in constrained situations have a greater chance of being useful. Also, investment in reusable frameworks can justify the investment in formal methods.

Formal verification of agile frameworks is not trivial. Errors in the framework core or library will affect all applications built with the framework. Although frameworks are meant to have stable structures in particular application domains, they are subject to change. Thus, formal verification of a single version of an agile framework may overlook important properties in this ever-changing context. How to gain effective comprehension within continuous evolution of agile software frameworks becomes more and more crucial for software practitioners.

In this paper, we present a method of applying model checking techniques to verify behavioral properties of evolving agile frameworks at an architectural level. Our hypothesis is that the obtained results can be used to understand, evaluate, and justify the maintenance activities in software evolution. Also, some important but implicit assumptions about the application domain of the framework can be identified by carrying out this rigorous approach.

Niu N. and Easterbrook S. (2005).
On the Use of Model Checking in Verification of Evolving Agile Software Frameworks: An Exploratory Case Study.
In *Proceedings of the 3rd International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems*, pages 115-117
DOI: 10.5220/0002560801150117
Copyright © SciTePress

2 Verification of Evolving Software Systems by Model Checking

Figure 1 shows the process model of our approach. Square-, round-, dashed-boxes, circle, and cylinder represent original inputs, abstracted models, hierarchical structures, external entity, and the knowledge base respectively. The output is the description of the model checking result (a proof or a counter-example) whereby software practitioners can use as a feedback for further study of the effects in software evolution. Given the input of the source code, a sequence of well-defined reverse engineering operations, such as filtering, componentizing, projecting, collapsing, can produce a structural model of the software system. To generate the behavioral model at a higher level of abstraction, more human input is needed.

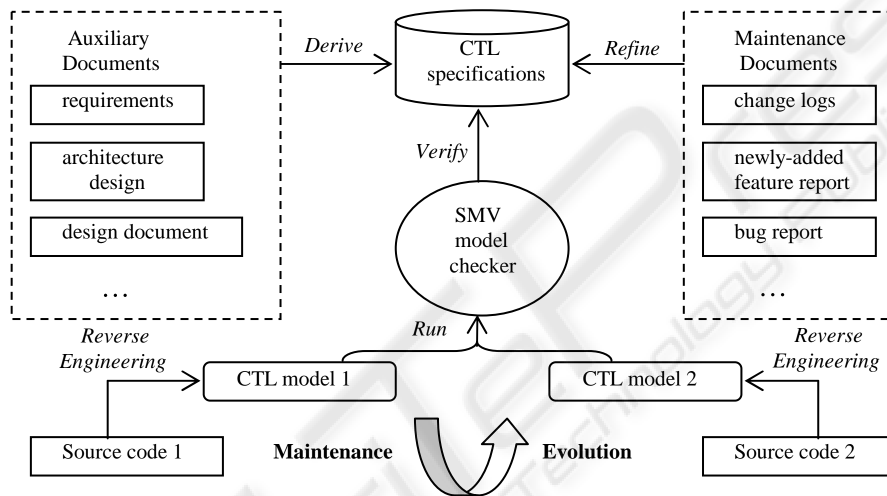

Fig. 1. Process model

3 Case Study

The subject software system of our case study is an industrially relevant, domain-specific, light-weight, database-centric Web application framework – "Prothos" [1]. "Light-weight" means that Web applications developed from Prothos should not have data-intensive or long-duration transactions. We focus on Prothos' *persistent object manager* (POM) (versions 4.1.3 and 4.1.4) that supports concurrent transactions based on consistent data management. The models extracted from the source code are compressed into one view shown in Figure 2. Prothos' optimistic-locking concurrency-control mechanism guarantees both serializability and deadlock freedom. However, POM 4.1.3 suffers from a "starvation" problem: One "heavy-weight" transaction that stays in *executing* for an arbitrary long time will keep others from successfully issuing exclusive-mode locks. This also challenges Prothos' specific domain assumption.

We develop the input of the CTL models based on a simple banking scenario: There are two accounts *A* and *B* with original balances of $100 and $200 respectively. Let Tr_1 and Tr_2 be two transactions where Tr_1 transfers $20 from *A* to *B*, and Tr_2 transfers $40 from *B* to *A*. The consistency constraint is that the total amount, *A* + *B*, should always be $300, which is expressed in ***SPEC 1***. If every transaction eventually ends in *committing* or *aborting* states after *starting*, the system is both deadlock free and starvation free. ***SPEC 2*** states this behavioral property in CTL. Both models shown in Figure 2 satisfy ***SPEC 1***. However, ***SPEC 2*** turns out to be false in POM 4.1.3's model. As mentioned earlier, this is caused by the starvation problem. After version 4.1.4 adds the new transition, ***SPEC 2*** is satisfied, which effectively verifies the corrective maintenance activities in software evolution. Since Prothos is purely an agile framework, a genuine Web application, Hermes [1], has been used to faithfully reflect our investigation and further show the validity of the case study.

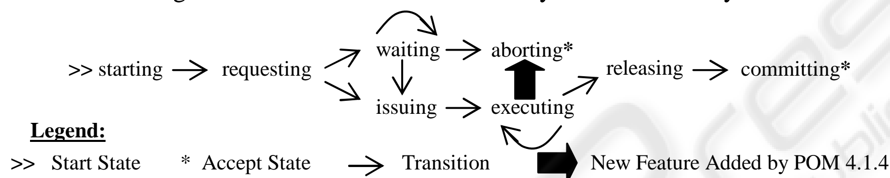

Fig. 2. State transition models

AG((A_value + B_value) = 300) ***(SPEC 1)***

AG((tr1.state=starting) → (AF (tr1.state=committing | tr1.state= aborting))) &

AG((tr2.state=starting) → (AF (tr2.state=committing | tr2.state= aborting))) ***(SPEC 2)***

We have proposed a model checking process to effectively verify evolving agile software frameworks by capturing behavioral changes at an architectural level. From our experience applying this approach in the case study, we feel that it has rich value in helping practitioners control software evolution and understand framework's application domain. More empirical studies need to be taken in a systematic way. Furthermore, techniques that permit round trip synchronization between models and source code deserve further investigation. Whether and how to leverage formal methods in software development and maintenance have been controversial for decades. Based on our work, we feel that formal methods can help software engineers gain some insightful comprehension about the system. But they are by no means a panacea.

Acknowledgments. We thank Jim Hoover and Kenny Wong for taking part in the study of Prothos and their extensive help in improving the presentation of this paper.

References

1. Niu, N.: Formally Understanding the Behavior of a Framework's Transaction Management. Master's Thesis, Department of Computing Science, University of Alberta, 2003.