Innovations Syst Softw Eng
DOI 10.1007/s11334-009-0114-3

ORIGINAL PAPER

An approach to applying SOFL for agile process and its application in developing a test support tool

Shaoying Liu

Received: 25 November 2009 / Accepted: 4 December 2009

Abstract Structured Object-Oriented Formal Language (SOFL) is a representative formal engineering method for software development. It offers a three-step specification approach to constructing formal specifications, and specification-based inspection and testing for verification and validation. In this paper, we describe a novel approach to applying the SOFL method to achieve agile development process. This approach results from our experience in several collaboration projects with industry, and aims to strike a balance between the fast delivery of software product and the assurance of its quality. We have tested the approach in developing a prototype test support tool.

Keywords Formal engineering methods · SOFL · Agile process · Semi-formal specification

1 Introduction

Formal methods have made significant contributions to software engineering by offering formal specification, refinement, and verification techniques for achieving correct programs. They are, however, "too good to be true" for most industrial software development projects. Formal methods can theoretically ensure that a program is correct with respect to a formal specification, but have no means to guarantee that the specification accurately and completely reflects the user's requirements. Although techniques for validating formal specifications, such as specification animation and testing [1,2], have been studied, they are only applicable to restricted specifications and their power is very limited. To enable formal methods to be effective in producing correct programs, they must be put in "good hands". According to the latest survey on formal methods practice in industry by Woodcock et al. [3] and our experience in Japan, such good hands are unfortunately very few in the industry. In Japan, only two teams of two companies have applied some formal methods, such as VDM or VDM++, in real projects (explicit operation specifications were mainly used) [4,5] in recent years. Education and training have been considered to be an effective way of promoting the adoption of formal methods in practice, but the reality is disappointing. According to our informal investigation, almost none of the students who received formal methods education at university and some company in Japan are directly using formal methods in their own industrial projects (but some may use them in mind to help their thinking, nobody knows). There are many technical difficulties and practical constraints in using formal methods in practice, but the most frequently heard comment from practitioners is "using formal methods is difficult and time-consuming".

Considering "too good to be true" to be very different from "too bad to be true", we believe that formal methods can play the role of "science", and effective technologies can be developed based on them. To reduce the complexity and improve the intuitivity of formal methods, we have been concentrating on developing *formal engineering methods* (FEM) over the last twenty years. By FEM we mean engineering methods showing how formal techniques can be effectively integrated into conventional software development techniques to improve the rigor, comprehensibility, and tool supportability of practical software development process. One of the significant results is the SOFL method [6,7] that offers a three-step specification approach to constructing formal specifications,

This work is supported in part by NII Collaborative Research Program and the NSFC Grant (No. 60910004).

S. Liu (✉)
Department of Computer Science, Hosei University, Tokyo, Japan
e-mail: sliu@hosei.ac.jp

Published online: 22 December 2009

and specification-based inspection and testing for verification and validation.

The three-step approach takes a natural process, consistent with the humans' evolutionary way of learning, and includes *informal*, *semi-formal* , and *formal specifications*, where the former two are used for capturing requirements while the latter is used for abstract design. The informal specification is basically written in natural language, but organized into three sections: functions, data resources, and constraints. The functions section describes a set of functions to be provided by the program and they can be organized in a hierarchical structure. The data resources section lists all data items necessary to be used in realizing the functions. The constraints section provides a collection of necessary constraints on either the functions or the data resources; they may include constraints in business policy, safety, security, availability, and/or efficiency. The overall goal of an informal specification is to achieve as complete coverage of requirements as possible. The semi-formal specification aims to group the related functions, data items, and constraints into modules, which are the essential units of a SOFL specification, to precisely define the data items using well-defined types, and to specify each of the functions as a process (operation) using pre- and post-conditions. The signature of each process is precisely defined, but its pre- and post-conditions may be described informally. To ensure the understandability, the informal pre- and post-conditions may be organized structurally. Based on the semi-formal specification, a formal design specification can be constructed. We emphasize the importance of the principle of first designing the architecture of the system and then writing formal textual specifications for all the components (e.g., processes, data flows, and data stores) involved in the architecture. The architecture is expressed using a formal graphical notation called *condition data flow diagram* (CDFD) and the process specification is written using a formal textual notation similar to VDM-SL. This way of constructing formal specifications strikes a good balance between visualization and formalization, thus reducing the complexity of formal specifications, improving its comprehensibility, and avoiding unnecessary workload in formal specification evolution.

Although SOFL has made useful progress in improving the practicality of formal methods, our experience in several joint projects with industry suggests that the SOFL method still faces a challenge in addressing the problem of how to quickly deliver good quality software products to clients. The major reason is that completing a formal design specification takes time, especially for a large-scale system. As we all know, nowadays the competition among software companies is intensive, requiring fast delivery (or demonstration) of even working products, increased functions, and the final product with acceptable quality. In Japan, more and more big companies tend to merely write a requirements specification and leave the task of design and implementation of the program to other smaller software companies. In such a situation, requiring either side to write formal specifications seems hardly practical.

In contrast, agile software development methods [8] seem to be more appropriate to deal with this situation, because they promote a disciplined project management process that encourages frequent inspection and adaptation, a leadership philosophy that encourages teamwork, self-organization and accountability. However, a potential risk of these methods is that the working products or final product is unlikely to have a well-designed structure and document. This will inevitably increase the difficulty and cost for evolution of the product, especially when the working product is getting bigger and more complex.

In this paper, we put forward a new approach to applying the SOFL method for agile development process while ensuring the quality of software. This approach aims to strike a balance between quick delivery of a product and the assurance of its quality. The essential idea of the approach includes the following points: (1) developing an informal requirements specification and then refining it into a complete semi-formal specification, (2) prioritizing the processes (defining functions to be implemented) to determine their order for implementation based on the importance and the client's interest, (3) writing a formal specification only to help understanding of ambiguous statements in the semi-formal specification (but paying no attention to its maintenance), (4) training developers in formal methods to strengthen their way of thinking [16], (5) implementing the process specifications in turn based on their priorities, and (6) applying specification-based inspection and testing for verification and validation of both working products and the completed product. This approach has been tested in our recent project for developing an automated tool support for specification-based testing. Part of this method has been tested by practitioners in the company receiving the influence of our research and education.

The rest of this paper is organized as follows. Section 2 details the proposed approach. Section 3 presents an example of building the test support tool to demonstrate the feasibility of our approach. Section 4 discusses the benefits and interesting issues of our approach. Section 5 compares our work with related work. Finally, in Sect. 6, we conclude the paper and point out future research directions.

2 Applying SOFL to achieve agile process

In this section, we concentrate on the discussion of all the points included in our approach to applying the SOFL method for agile development process.

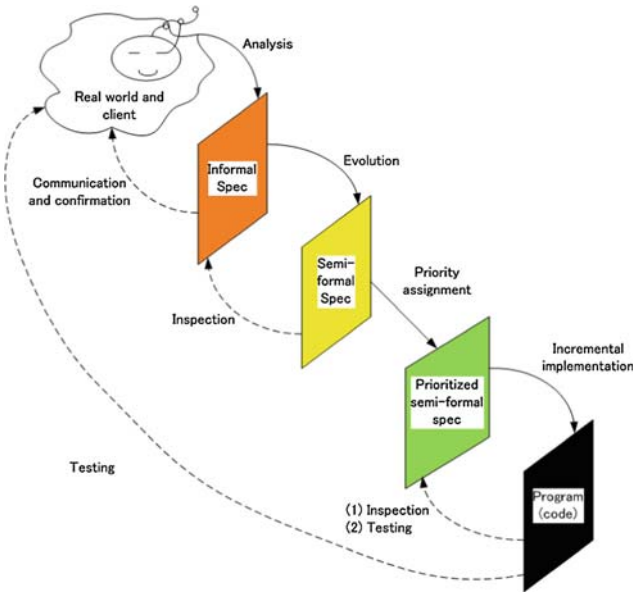

Fig. 1 Agile process of applying the SOFL method

Figure 1 shows the agile process of applying the SOFL method, which illustrates how the related activities are arranged and connected. The underlying principle of this process is first to achieve a complete semi-formal specification and then to carry out an incremental implementation (possibly including formal design). It also supports specification-based inspection and testing as the verification and validation techniques to ensure the quality of the documents produced.

To prevent frequent and considerable changes during the evolution of the potential program, it is necessary to understand rather precisely what to be implemented in the early phases of the development, before the implementation actually starts. To fulfill this goal, a semi-formal specification needs to be achieved, based on an informal specification. As briefly explained in Sect. 1, the informal specification aims to obtain a complete coverage of requirements at informal level, while the semi-formal specification is intended to achieve the precise and deeper understanding of the informal requirements, and to organize the related items into modules. In particular, all data items must be declared with well-defined data types, and all required functions must be specified as process (operation) specifications using pre- and post-conditions in a semi-formal manner. The signature of the processes is formally defined but their pre- and post-conditions may be described informally. The formal declaration of the data items can significantly help the developer to deepen his or her understanding of the requirements and to determine the signature of processes. Precisely defining the signature of a process is the first step of specifying the function of the process. Keeping the pre- and post-conditions informal ensures a good readability of the process specification for all the involved members. In fact, according to our industrial partner, the semi-formal specification is much more popular than formal specifications in practice.

To allow for the fast delivery of working products to the client for communication, the processes specified in the semi-formal specification must be prioritized to facilitate an incremental implementation from more important functions to less important functions. In order to ensure a quality implementation of a process specification, a correct and detailed understanding of it is crucial. This can be achieved through constructing a formal specification, but since writing a formal specification usually takes time, it can only be applied to clarify ambiguities; no maintenance of formal specifications is required.

During the entire development process, specification-based inspection can be applied to ensure the consistency between a high level document and the low level document developed from it. For example, inspection is applied to check the consistency between the informal specification and the semi-formal specification, between the prioritized semi-formal specification and the program. The underlying principle of such an inspection is to check whether every item, which is a data item, function, or constraint, in the high level document is properly implemented in the low level document. These inspections can ensure that all requirements defined in the high level document are properly realized in the low level document. Furthermore, to find more bugs and to validate the program (and the corresponding specification), specification-based testing can be carried out. Test cases can be generated based on the semi-formal specification. The developer and the client can work together to detect bugs through testing.

To ensure a fast but quality delivery, without causing unnecessary and frequent changes in the working product, the cycle of incremental implementation, inspection, and testing can be carried out in an agile manner (e.g., by applying the Extreme Programming techniques [9]). Using this agile cycle, together with a well-established semi-formal specification in the early phase, the whole development will always have a good traceability and will not fall into chaos, thus ensuring a fast and quality delivery of both working products and the final product.

3 An example

Building a software tool support for automatic specification-based testing is one of the major tasks for our joint project with NII and industry. Although this is a research project, our industry partner expects us to complete the tool as early as possible. In this sense, the project has no big difference from commercial software projects.

```
Informal Specification of the Test Support Tool

1. Functions
(1) Automatic derivation of functional scenario form (FSF) from a
    SOFL process specification.
  (1.1) Transformation of compound expressions.
  (1.2) Transformation of quantified expressions.
  (1.3) Transformation of predicate expressions into FSF.
(2) Automatic test case generation based on functional scenarios.
  (2.1) Test case generation from disjunctions.
  (2.2) Test case generation from conjunctions.
  (2.3) Test case generation from relations.
   ...
(3) Automatic mapping the test cases generated to Java expressions.
(4) Automatic execution of the program with test cases.
(5) Automatic collection of program execution results.
(6) Automatic test result analysis for bug detection.
(7) ···
2. Data resources
(1) Specification file
(2) Program file
(3) Test cases file
···
3. Constraints
(1) More than two test cases must be generated based on a single
    functional scenario.
(2) ...
```

Fig. 2 Informal specification of the test support tool

We took the agile approach described in this paper to develop the system, in order to ensure a fast delivery of both the working product and the final product. The first step is to carry out the requirements analysis and complete an informal specification, as shown in Fig. 2. More than six functions are listed, and some of them, such as function (1) and (2), are decomposed into sub-functions. In addition, over three data items are given in the data resource section, and more than one constraint is documented.

To clarify the meaning of the statements in the informal specification, we developed it into a semi-formal specification, as shown in Fig. 3. The partial information of the two modules, *SOFL_Parser* and *AutoTestCaseGeneration*, is presented (all other modules are omitted for brevity). In each module specification, necessary types are defined, such as *TypeTable* and *DataType* in the first module and *TestCase* and *Relation* in the second module; related processes are defined, such as *Parser* in the first module and *GenerateTestCaseFromRelation* in the second module. Each process specification has precise signature, but its pre- and post-conditions are written informally.

On the basis of this semi-formal specification, we assign a priority to each module. Since the SOFL_Parser module is concerned with the fundamental functions that are the basis for realizing other functions such as automatic test case generation, we start an incremental implementation from this module, and apply the small cycle of implementation, inspection, and testing to each implemented program.

```
Semi-formal specification of the test support tool

module SOFL_Parser;

type

TypeTable = set of DataType;  /* ···*/
DataType = composed of
      type_name: TypeName
      element_type: seq of DataType
      end;

TypeName = string;
···

process Parser()
ext rd operation_spec_file
  wr type_table
  wr operation_table
post (1) Derive a functional scenario form from the operation_spec_file.
      (2) Generate the type-table and operation-table from the operation_spec_file.
          ...
end_process;

Module AutoTestCaseGeneration;
 type
  TestCase = seq of Variable * Value;
  Relation = composed of
               left: Expression
              op: RelationalOperator
              right: Expression
  ···
 var
  test_set: set of TestCase;
  ···
 process GenerateTestCaseFromRelation(rel: Relation)
 ext wr test_set: set of TestCase
 post  (1) Generate test cases for all the variables in both rel.left and rel.right to
           meet the relation.
       (2) Add the generated test cases to test_set.
       (3) Continue the generation until a termination condition is met.
 end_process;
 ...
end_module;
...
```

Fig. 3 Semi-formal specification of the test support tool

Such a way allows us to see the new progress and to evaluate whether the new progress is in the correct direction. Therefore, the risk of failure of the whole project is small. This project is still going on, but as the latest result, we have achieved the following functions:

- Specification analysis and transformation to an functional scenario form (FSF). To prepare for automatic test case generation from a specification, it is essential to analyze the specification and to transform it into an FSF. The analysis of the specification aims to ensure its syntactical correctness and semantic consistency, which will set up a foundation for correctly transforming the specification into an equivalent FSF. The transformation is implemented based on the algorithm that was presented in our previous publication [10].
- Test case generation. On the basis of each functional scenario of the FSF, the tool supports the application of existing algorithms to generate test cases. We have implemented only part of the data types available in the SOFL language so far, which are numeric types (e.g., natural numbers, integers, and real numbers), some set types (e.g., *set of int*), and some sequence types (e.g., *seq*

of int). The implementation of composite types and map types are not completed yet. Currently, test cases are generated by the user, with the tool support, but the future development of the tool will lead to automatic test case generation.
- Construction of *driving module*. To run the program with a test case, the tool automatically creates a driving module (a class method in Java) that will take a test case as input and call the program to be tested. It will also automatically collect the test result after the execution terminates for test result analysis.
- Test result analysis. With the generated test case and actual test result, the tool will automatically evaluate the pre- and post-conditions, and determine whether the test has detected a bug in the program.

What we have achieved so far in the tool construction are the basic functions; there is a long way to go before reaching a really mature tool for practical use. The experience we have gained, however, has set up a foundation and direction for further tool development in the future.

4 Discussion

Through our experience in the ongoing project, we have found several benefits of our approach, and at the same time found some interesting issues that need to be further investigated.

The major benefits of the approach include

- The progress of the software tool construction can be quickly delivered for evaluation, discussion, and improvement. Compared to our previous experiences in using formal methods, this benefit is significant. We believe that this benefit is extremely important for industry because frequent communications between the client and the developer can be realized to ensure that the client's changing requirements can be timely incorporated into the new version of the tool.
- The traceability between documents is good and helpful in ensuring a systematic progress in the implementation cycle (i.e., implementation, inspection, and testing). Since a semi-formal specification is constructed for almost the entire system before the implementation starts, it provides a rather stable basis for implementation. Although some changes of the originally defined functions in the semi-formal specification may take place during implementation, they are relatively few.
- The program structure does not go through big operations as the scale of the implementation grows. Our experience suggests that this strong point mainly benefits from the well-constructed semi-formal specification.
- Using formal specification techniques in mind strengthens our way of thinking when writing documents. Although in the proposed approach we do not emphasize the necessity of directly using formal specification techniques, we advocate strongly the role of formal methods education. In our tool building project, we taught the SOFL specification language to all the members in advance, and encouraged them to apply it to resolve ambiguities and to deepen their understanding of requirements whenever necessary and possible. As a result, few members wrote formal specifications (one tried in the beginning, but stopped after a while for the sake of difficulty and time). Of course, some functions, such as GUI and diagram drawing, may not be suitable for formal specification in terms of pre- and post-conditions.

On the other hand, without systematically using formal specifications, we have encountered the following challenges:

- Program inspection is not necessarily rigorous. Since the pre- and post-conditions of a process specification are expressed informally, the derivation of precise questions for a checklist for inspection of the program can be difficult. In our case, experience became crucial.
- Program testing is not necessarily rigorous. For the same reason as above, the testing of the program on the basis of the semi-formal specification cannot be rigorously carried out. This includes test case generation from the specification and test results analysis based on the specification. In our project, most tests were carried out based only on the signature of processes and the developers' experience.

There seem to have many challenging issues in addressing the conflict between quality and fast delivery of software products. Our approach presented in this paper can be more flexibly used to deal with this problem. If time allows, a formal design specification can be constructed first and then an incremental implementation cycle follows. If time is limited, the implementation cycle can be carried out on the basis of the semi-formal specification.

5 Comparison with related work

Looking at existing work reported in the literature, we believe that our approach to combining the SOFL formal engineering method with the essential principle of agile methods is quite unique. In general, agile methods, such as Extreme Programming (XP) [9], Scrum [11], and Dynamic Systems Development Method (DSDM) [12], focus on strategies for software

development management, emphasizing human interactions (between developers and between the developer and the client) and performing programming, inspection, and testing in small cycles. The development process is evolutionary and incremental, and does not advocate documentation for the sake of time. In our approach, not only managerial strategies and human interactions are also emphasized during the entire process of the development, but documentation for informal and semi-formal specifications are emphasized to overcome the weakness of existing agile methods: evolution of the system is getting more difficult and time-consuming when it is becoming bigger and more complex.

As reported by Black et al. in [13], some researchers have proposed to integrate the techniques of formal methods into agile development process in order to achieve a balance between quality and fast delivery of software products, although few of them have actually shown the compromise necessary for adapting the formal techniques to agile process. eXtreme Formal Modeling [14] is considered to be an agile method that advocates the construction of a formal model based on a natural language description and the use of model checking to check whether the formal model satisfies desired properties expressed in linear temporal logic. XFun method [15] combines X-machines with the unified process to allow the system to be built with the assurance of its correctness with respect to the given user requirements. The requirements are translated into X-machines that are verified with model checking technology to determine whether various safety properties hold. Compared to these approaches, our agile method makes more but necessary compromise in the aspect of adopting formal specification and formal verification techniques. Our experience in applying formal specification and specification-based verification techniques to various small systems development has convinced us that writing formal specification is an effective way to help the developer understand and even explore requirements, but their maintenance to reflect the frequent changes during the system development is a barrier for fast delivery, and in most cases such a maintenance is rarely possible. Specification-based verification, including model checking and formal proof, is almost out of question in practice for rich typed languages (for both specification and programming), such as VDM, Z, and Java. That is why we only advocate the use of formal specification as a means to help the developer deepen his or her understanding of the requirements without the need of maintaining the specification, and the use of inspection and testing rather than model checking and formal proof for verification and validation. We emphasize the importance of completing a semi-formal specification before starting implementation. This principle is important because the semi-formal specification serves as a firm foundation for subsequent development, which prevents frequent functional and architectural changes during implementation and verification activities. But for the development of a large system, building a complete semi-formal specification may also take a long time during which no working products or part of the final product could be demonstrated to the client. How to solve this problem still remains as a challenge to our method proposed in this paper. We are currently making effort to explore the possibility and effectiveness of combining prototyping and the SOFL three-step modeling approach, which we believe will lead to an effective solution and tool development in the future.

6 Conclusion and future work

In this paper we present an approach to applying the SOFL method to achieve agile development process. It strikes a balance between software quality and fast delivery. The quality can be ensured by a semi-formal specification for the entire system, which will serve as a foundation for implementation, inspection, and testing. The fast delivery of working products and the final product can be realized by an incremental implementation cycle based on the prioritized semi-formal specification. We have tested the approach in our ongoing joint project with industry for building a test support tool, and found several benefits and some weaknesses.

In the future, we are interested in investigating the issues of how to integrate prototyping into our approach described in this paper for quality agile software development, and in developing effective tool support for the new method.

References

1. Miller T, Strooper P (2002) Model-based specification animation using testgraphs. In: Proceedings of 4th international conference on formal engineering methods. LNCS, Springer, Shanghai, pp 192–203
2. Liu S (1999) Verifying consistency and validity of formal specifications by testing. In: Proceedings of the world congress on formal methods in the development of computing systems. LNCS. Springer, Toulouse, pp 896–914
3. Woodcock J, Larsen PG, Bicarregui J, Fitzgerald J (2009) Formal methods: practice and experience. ACM Comput Surv 41(4):1–39
4. Kurita T, Nakatsugawa Y, Ohta Y (2005) Applying formal specification method in the development of an embedded mobile FeliCa IC Chip (in Japanese). In: Proceedings of the 2005 software symposium. Toyama, Japan, pp 73–80
5. Sahara S (2004) An experience of applying formal method on a large business application (in Japanese). In: Proceedings of 2004 symposium of science and technology on system verification, AIST, Osaka, Japan, pp 93–100
6. Liu S, Offutt AJ, Ho-Stuart C, Sun Y, Ohba M (1998) SOFL: a formal engineering methodology for industrial applications. IEEE Trans Softw Eng 24(1):337–344
7. Liu S (2004) Formal engineering for industrial software development using the SOFL method. Springer, Berlin

8. Abrahamsson P, Salo O, Ronkainen J, Warsta J (2002) Agile software development methods: review and analysis. Espoo 2002, VTT Publications 478
9. Beck K (1999) Embracing change with extreme programming. IEEE Comput 32(10):70–77
10. Liu S (2007) Integrating specification-based review and testing for detecting errors in programs. In: 9th international conference on formal engineering methods. LNCS, vol 4789. Springer, Boca Raton, pp 136–150
11. Schwaber K, Beedle M (2002) Agile software development with scrum. Prentice-Hall, Upper Saddle River
12. Stapleton J (1997) Dynamic systems development method—the method in practice. Addison Wesley, Reading
13. Black S, Boca PP, Bowen JP, Gorman J, Hinchey M (2009) Formal versus agile: survival of the fitest? IEEE Comput 42(9):37–45
14. Suhaib SM, Mathaikutty DA, Shukla SK, Berner D (2005) XFM: an incremental methodology for developing formal models. ACM Trans Des Autom Electron Syst 10(4):589–609
15. Eleftherakis G, Cowling AJ (2003) An agile formal development methodology. In: Proceedings of 1st South East European workshop on formal methods (SEEFM03), Hawaii, November 2003
16. Liu S, Takahashi K, Hayashi T, Nakayama T (2009) Teaching formal methods in the context of software engineering. Inroads SIGCSE Bull 41(2):17–23