Hermeneutics Framework: Integration of Design Rationale and Optimizing Software Modules

Mehmet Aksit
University of Twente
Enschede, the Netherlands
m.aksit@utwente.nl
and
Somayeh Malakuti
Software Technology group
Technical University of Dresden, Germany
somayeh.malakuti@tu-dresden.de

Abstract: - To tackle the evolution challenges of adaptive systems, this paper argues on the necessity of hermeneutic approaches that help to avoid too early elimination of design alternatives. This visionary paper proposes the Hermeneutics Framework, which computationally integrates a design rationale management system, an auto-adaptive control system and a reflective and modular event-driven language runtime together. The Hermeneutics Framework is, among others, suitable for implementing dynamic adaptive software systems that undergo intensive evolution cycles.

Key-Words: - hermeneutics, hermeneutics framework, design rationale, software evolution, self-adaptation, design alternatives, event-based languages

1 Introduction

Despite the abundant availability of design methods and programming languages, the quality of software still largely depends on the experiences and skills of software engineers; creating bug-free software while satisfying the desired non-functional requirements is considered as an extremely hard task, if ever possible.

Since several decades, a large amount of research has been carried out in many disciplines of software engineering, in requirement analysis, architecture design, design rationale management, patterns and styles, domain specific and general purpose languages, formalization of software from different viewpoints, automated testing and debugging etc. Although in each of these areas much have been accomplished, due to the lack of holistic approaches, the obtained benefits of the new technologies have remained limited.

These enormous challenges are somewhat understandable. First, the underlying theories show that the required algorithms for creating computational solutions to tackle many of software engineering challenges cannot be practical for general cases. For example, to satisfy arbitrary requirements within the following problem areas, automated program synthesis, allocation and clustering of computational elements, resolution of certain logical equations, satisfying multi-criteria constraints are either undecidable and/or show exponential time/space characteristics with respect to the number of parameters [12]. Second, the application domains of software systems are growing since software is now applied in many different disciplines. Third, the ever increasing speed of processors, parallel architectures, memory capacities and growing trends in different kinds of networked solutions, enable software engineers to realize increasingly powerful and complex software systems. Last but not least, the fragmentation of the software engineering topics into many sub-domains has made it extremely difficult for the researchers to propose holistic solutions; it is a challenge to gain an overview of the relevant disciplines and to make scientific publications which incorporate techniques from different sub-domains.

Nevertheless, in each sub-domain, there is a convergence in the proposed solutions; separation of concerns, proper modelling, and computable evaluation of models have become important. Furthermore, for most practical cases, the complexity of the required algorithms can be managed by restricting the design spaces with domain-specific semantic information and heuristics.

Based on our extensive research in various sub-domains of software engineering, this paper proposes a novel framework termed as

Hermeneutics Framework, for the holistic integration of design rationale management system, an auto-adaptive control system and a reflective and modular event-driven language runtime together.

This paper is organized as follows: Section 2 explains the problem with current software design processes; Section 3 explains the Hermeneutics Framework, and Section 4 outlines the discussions.

2 Problem Statement

The principle of separation of concerns is considered important in achieving the desired quality attributes in software [3]. Separation of concerns starts at the semantic level; semantics determine the elements that constitute the meaning of programs and specify the relationships among them. One may not expect software engineers to separate concerns at the programming language level any better than their understanding of the separation of concerns at the semantics level.

Consider for example, the specification of a program as shown in Fig. 1; the parts (a) and (b) depict the control-flow and data-flow of a simple program, respectively. f1, f2 and f3 are the functions and d1 and d2 and d3 are the data values exchanged between the functions. The function f1 initializes d1 with x, and the functions f1 and f2 increment this data value by 1, consequently. Depending on the interest, in principle every element in Fig. 1 can be considered as a concern: functions, data values, relationships among these and the implementation of the functions.

While implementing the semantics of the program shown in Fig. 1, software engineers must map the elements of the semantic model to the elements of the adopted programming language. However, there may be many factors that play a role in this context, such as the availability of the tools, experience of software engineers, efficiency of the generated code, etc. Even if the implementation language is fixed, there are still many mapping possibilities.

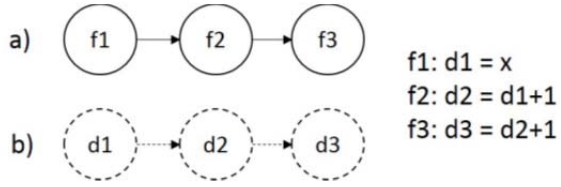

Fig. 1 a) Control-flow and b) data-flow diagrams of the illustrative example

Consider for example, some possible object-oriented implementations of our illustrative example as shown in Fig. 2.

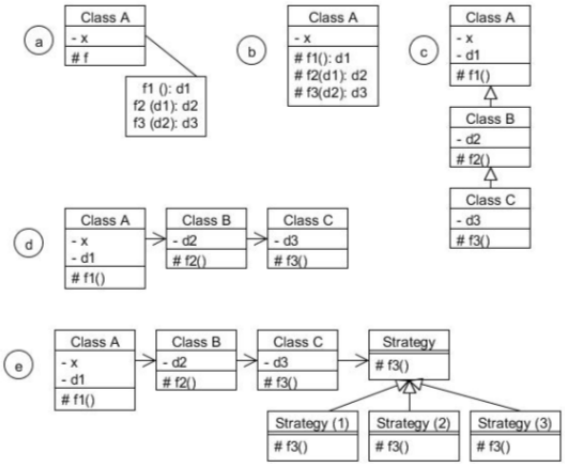

Fig. 2 Various object-oriented implementation possibilities of the program specified in Fig. 1

Although all these implementations have the same functional semantics, they have different quality characteristics. The implementation shown in (a) is expected to be faster because all the functions are in-lined. On the other hand, the concerns are not separated and therefore they are not separately reusable. In b), the functions are separated but tangled in one class. The relationship between the functions is realized through self-calls. Nevertheless, each of these functions can be overridden through inheritance. For example, it is assumed that in (c), the functions in classes B and C are implemented by invoking the corresponding function in their super class and incrementing the results by 1; in (d), each function is defined as a method in a separate class. This provides a clear separation as such enables each function be reused separately.

Due to time performance, memory performance or portability reasons, sometimes the implementation of a function may be replaced at runtime, although its semantics remain the same. In this case, the Strategy pattern [5] can be used. In (e), the function f3 is realized in this way.

There are several important issues in expressing the semantics of programs. First of all, there may not be even an explicit semantic model, making it very difficult to identify what the essential concerns are. Secondly, in practice, mapping the elements of a semantic model to the adopted language model is carried out informally, based on the intuition of software engineers and possibly some informal heuristics of a design method. Since there are many

alternatives in a mapping process, it is very hard to find out the best alternative. Thirdly, binding the elements of a semantic model to the elements of a programming language is in general carried out too early in the design process. Most methods start with the 'object identification' process where the decomposition of a program is determined more or less right in the beginning of the design process.

As shown by several publications [6], too early binding does not only hinder considering alternative mappings in later stages, but also results in information loss. Fourthly, the deterministic nature of the design models (such as the UML) and the computation model of the programming languages, force software engineers to resolve all the possible ambiguities before the design model and/or program is constructed. This is also the reason why software engineers are forced to make too early decisions along the software development process. Finally, due to changes in the context of the application or requirements, even a perfect mapping from a semantics model to a language realization may cease to become imperfect in the follow-up releases of program.

3 Hermeneutics Framework

To overcome the previously mentioned problems, we have been researching on a software engineering development environment called the Hermeneutics Framework, which facilitates holistic integration of three important stages in software design: design process, software construction/programming process, and the quality optimization process.

In traditional computer science, software is interpreted by a processor and/or transformed by a compiler to a form that is interpretable by a processor. Inspired from the hermeneutics philosophy [17], in this paper the term hermeneutics software is defined as the interpretation of software within the intentions of its creator(s). The conceptual architecture of the Hermeneutics Framework is shown in Fig. 3.

The Hermeneutics Framework system is a fuzzy-probabilistic reasoning system, which receives application requirements, design heuristics expressed as fuzzy-probabilistic rules, contextual rules, and the model of the target language as input. Here, the fuzzy-probabilistic heuristics refer to various application-specific and general-purpose rule libraries which can be extended, if necessary. Due to the fuzzy probabilistic nature, the rules can cope with logical and time related uncertainties.

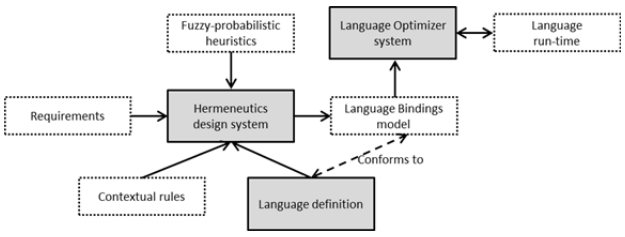

Fig. 3 Conceptual architecture of the Hermeneutics Framework

The contextual rules refers to the design rationale about the application context of the software to be designed; these rules can be, for example, assumptions about the users, deployment context and the related non-functional qualities. The model of the target language specifies the first-class abstractions of the language, which can be adopted to implement the application requirements.

As its output, the design system generates a fuzzy-probabilistic design model of the software solution, which is termed as Language Bindings model. This model is an instance of the target language model, in which each language element is augmented with a dedicated specification consisting of a *when* and a *how* part. The *when* part is expressed in a fuzzy-logic based notation and determines the relevancy of the element in the generated design. The *how* part specifies the way of realizing that element. For example, in the rule depicted in Fig. 4, RELEVANCE defines a fuzzy set, the IF and the THEN parts define the when and how parts of the specification, respectively.

```
<RELEVANCE> IF <alternative functionalities> &
              <implementations are algorithmic>
            THEN <adopt Strategy pattern binding>
```

Fig. 4 An example fuzzy-logic based mapping model

To derive the fuzzy quantifiers of the rules, the Hermeneutics design system gathers information about the desired quality requirements of the application and evaluates the relevance values of the rules accordingly. As such depending on the requirements, the relevancy of some rules can be increased or decreased. For example, if performance is important, all rules that refer to efficient implementations (such as in-lining) can be emphasized by increasing the relevance values of these rules. To simplify the binding process, a

threshold can be set if necessary; in such a case, if relevance drops under a certain value, it can be removed.

4 The Language Model

Depending on the adopted language paradigm, at the realization level, the concerns are represented and separated by one or more modules or by the elements that constitute to the modules of the programming language. Modules are assumed to be the first-class abstractions of a given language. For example, in object-oriented, functional and logical languages, first class abstractions are objects, functions or predicates.

While realizing the semantics model, software engineers should decide which element of the language model should represent a particular concern in a semantic model. This implies that the target element must be expressive enough to represent the intended semantics of the concern.

Currently, we are investigating event-driven language models that are expressive and flexible enough to represent a large category of semantics concerns.

The Language Optimizer system is an adaptive feedback control loop, which can be applied at design time or at runtime. In case of design time optimization, the system receives the Language Bindings model and the application requirements input, undertakes a defuzzification step, and generates an optimal design that fulfills the requirements. Since usually multiple quality attributes, such as performance and reusability are desired to be fulfilled, the system undertakes a multi-objective optimization technique to derive the design.

In case of runtime optimization, the optimizer system keeps the Language Bindings model and applies the optimization and defuzzification steps at runtime as incremental control steps.

5 Discussion

The proposed framework helps to avoid too early elimination of the design alternatives, because the design rationale system has a fuzzy-probabilistic nature and is computationally integrated with the design/programming models. The Language model provides the necessary adaptability through event-driven modular reflection. The language environment is tailored to couple it with the design rationale system and the optimizer. The optimizer is a multi-objective optimization system with the necessary defuzzification algorithms.

The paper published by [18] proposes a similar approach as ours. This paper presents a framework that aims to discover the optimum architecture solutions within design space. This framework adopts Object Process Methodology (OPM), Colored Petri Net (CPN) and feature model. Our approach is different from the presented approach in the following ways: First, we propose a fuzzy-probabilistic design and optimization system which can cope with uncertainties, as such it eliminates too early elimination of the alternative designs. Second, our optimizer supports design time and run-time optimization techniques, whereas the system proposed by [18] is restricted by design time optimization. Finally, our language system is based on modular reflective language which is capable of implementing event-driven, object-oriented and aspect-oriented solutions. Whereas in [18], solution are limited to component architectures.

The Hermeneutics Framework is, among others, suitable to implement self-adaptive software systems [1]. Such software systems must adapt themselves to the changes in the application requirements and/or contextual rules; multi-objective optimization is usually required to achieve a design that fulfills multiple quality requirements.

During past years, we have been investigating various underlying techniques and models for the Hermeneutics Framework. We have developed a design rationale system [15, 13, 7, 16], event-driven language system [8, 9, 10, 12], and the optimization system [14, 4]. We are currently investigating this framework as a means to holistically integrate these techniques.

References:

[1] M. Aksit and Z. Choukair. Dynamic Adaptive and Reconfigurable Systems Overview and Prospective Vision. In Workshop on Distributed Auto-adaptive Reconfigurable Systems (DARES). IEEE Computer Society, 2003.

[2] L. M. J. Bergmans, W. K. Havinga, and M. Aksit First-Class Compositions–Defining and Composing Object and Aspect Compositions with First-Class Operators. TAOSD, 2012.

[3] E. W. Dijkstra. EWD 447: On the role of scientific thought. Selected Writings on Computing: A Personal Perspective, pages 60–66, 1982.

[4] A. J. de Roo, H. Sözer, and M. Aksit. Composing Domain-Specific Physical Models

with General-Purpose Software Modules in Embedded Control Software. Software and Systems Modeling, online pre-publication, 2013.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1994.

[6] F. Marcelloni and M. Aksit. Automating Software Development Process using Fuzzy Logic. In E. Damiani, C. L. Jain, and M. Madravio, editors, Soft Computing in Software Engineering Series: Studies in Fuzziness and Soft Computing, volume 159. Springer,2004.

[7] F. Marcelloni and M. Aksit. Fuzzy Logic-based Object-Oriented Methods to Reduce Quantization Error and Contextual Bias Problems in Software Development. Fuzzy Sets and Systems, 145(1), 2004.

[8] S. Malakuti and M. Aksit. Event Modules: Modularizing Domain-Specific Crosscutting RV Concerns. In TAOSD, Lecture Notes in Computer Science, Volume 8400, 2014, pp 27-69.

[9] S. Malakuti and Mehmet Aksit. Event-Based Modularization of Reactive Systems. In Concurrent Objects and Beyond, Volume 8665, 2014, pp 367-407.

[10] S. Malakuti and Mehmet Aksit. Emergent Gummy Modules: Modular Representation of Emergent Behavior. In 13th Generative Programming: Concepts and Experiences (GPCE), ACM, 2014 .

[11] S. Malakuti. Specification of the GummyModule Language. Technical Report TR-CTIT-12-31, December 2012.

[12] O. Maimon and D. Braha. On the complexity of the design synthesis problem. Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on, 26(1):142–151, 1996.

[13] J. A. R. Noppen, P. M. van den Broek, and M. Aksit. Software Development with Imperfect Information. Soft Computing - A Fusion of Foundations, Methodologies and Applications, 12(1), 2008.

[14] H. Sözer, B. Tekinerdogan, and M. Aksit. Optimizing Decomposition of Software Architecture for Local Recovery. Software Quality Journal, 21(2), 2013.

[15] B. Tekinerdogan and M. Aksit. Synthesis-Based Software Architecture Design. In M. Aksit, editor, Software Architectures and Component Technology. Kluwer Academic Publishers, 2001.

[16] B. Tekinerdogan and M. Aksit. A Comparative Analysis of Software Engineering with Mature Engineering Disciplines Using a Problem-Solving Perspective. In A. H. Dogru and V. Bicer, editors, Modern Software Engineering Concepts and Practices: Advanced Approaches. Information Science Reference, Hershey, 2011.

[17] R. William L. Dictionary of Philosophy and Religion. Sussex: Harvester Press. p. 221. ISBN 0855271477, 1980.

[18] Renzhong Wang and Cihan H. Dagli. Computational system architecture development using a holistic modeling approach. Procedia Computer Science, 12(0), 2012.