1992

The Management of Uncertainty in Software Development

Luqi; Cooke, D.

IEEE

THE MANAGEMENT OF UNCERTAINTY IN SOFTWARE DEVELOPMENT

BY
LUQI, NAVAL POSTGRADUATE SCHOOL, MONTEREY, CA.
D. COOKE, UT EL PASO, EL PASO, TX.

Abstract

A high level view of the uncertainty in specification is presented to explain the problems inherent in software development and maintenance. The view supports the need for prototyping throughout software development and mainte nance.

1. Introduction

Software maintenance accounts for more than half of the total software cost. Each time a new software system is put into use, some fraction of the work force must be devoted to its maintenance. If it is assumed that all systems require some maintenance effort and a constant work force engages in software development for a long time, the fraction of effort available for developing new systems will get small, and can be kept from vanishing only by retiring or replacing some old systems [Mills]. There has been a great deal of interest in reducing maintenance costs. Prototyping is one promising approach to this goal [Boehm].

The prototyping of a system to be built is motivated by a desire to manage the uncertainty in developing the system. Management of uncertainty is an important issue in both software development and maintenance. There are at least two different kinds of uncertainty in software development. Both are centered around software specifica tion.

The first type has to do with the uncertainty as to whether or not a given description is truly a specifica tion of the software to be developed. In many past software development efforts, little has been done to formally make this decision which lies at the heart of requirement validation. The failure to make this determination may result in a great deal of wasted effort and money. It has been observed in many software projects that the validation of software specifications is typically completed during maintenance.

Large amounts of money are spent each year designing, coding, and testing soft ware products according to incorrect specifications. It is well known that the earlier corrections are made to specifications, the less the corrections cost. Rapid prototyping offers a framework within which software validation can be accomplished before these costs are incurred.

The second type of uncertainty has to do with the lifetime of a valid specification. According to Lehman [Leh] there are two types of programs: those based upon specifications which are valid forever and ones which are based upon specifications which may change over time. Most programs have parts that are a mixture of the two types. Lehman's work suggests that a validated specification (i.e., a specification which we are certain is a specification) may be axiomatic (i.e., fixed) or nonaxiomatic (i.e., likely to change eventually). From a more general point of view, there is a need for a periodic assessment of the validity of nonaxiomatic specifications over the life of a software product.

In this paper a section is devoted to each type of uncertainty in software specification. Then the reader is introduced to the prototyping framework of CAPS so that he/she can observe how prototyping can help the designer manage uncertainty.

2. The Uncertainty in Software Validation

In the current software development environment, after completing requirements analysis, the user "signs-off" on the resulting specification. The signing-off of the user implies that the requirements have been validated. Thus, there is no problem of conscience in hiding the software engineers from the user while the product is built. The only responsibility of the software engineer is to show that the product is correct with respect to the "validated" requirements. The process of demonstrating that software is correct is called verification, and it is typically accomplished by testing the programs with respect to the "validated" require ments.

In reality, programs are typically verified with respect to invalid requirements. This statement is supported by maintenance curves which depict the number of changes made to software over time.

Such a curve is given in figure 1. Notice that immediately after the installation of each version of a system, the number of changes made to the software is quite high. There are three types of changes made to the software: the corrective change (which is 20% of the total number of changes to the delivered software), the adaptive change (25%), and perfective changes (55%). [Dun]

Fig. 1. Number of Corrections to Software

The corrective changes are due to logic errors which were missed during verification. The perfective change is performed to make the system conform better to the user's requirements and reflects the inadequacy of validation. The purpose of an adaptive change is to adapt the system to a changing environment. Adaptive changes are responses to requirements changes which can be planned or unplanned. Unplanned changes are the most expensive kinds of changes. The adaptive changes in the maintenance phase correspond precisely to the management of the nonaxiomatic specifications upon which a system is partially based.

The high incidence of perfective changes supports the idea that currently software engineers actually validate requirements during the early stages of maintenance. The goal of validation is to reduce the ambiguity in specification and the uncertainty that the specification captures the users' intentions.

3. The Uncertainty in Software Evolution

A model of a specification is a program which satisfies the specification. For a given specification, there may exist many models. As a system evolves, the validity of some specifications may change. We call these specifications nonaxiomatic [Ram]. Nonaxiomatic specifications result in Lehman's E-Type programs.[Leh] When the truth of a specification changes, it is possible that the set of models for the specifications may change.

Consider a specification of a program represented by a specified relation R*, and a computer program represented by a computed relation R, where partial correctness of the computer program requires R* to contain R. There are two fundamental types of changes: A change may **delete** models of a specification (the extreme case resulting in inconsistency); or a change may **add** models of a specification (the extreme case being a tautology).

Examples of each case follow. Suppose a First Order Logic specification is: (All x Exists y) (nat(x) & y = f(x)). Here nat is a relation which is true when x is instantiated with a natural number. In this case the specified relation R* contains the tuples <1,f(1)>, <2,f(2)>, ...

One model of this specification is: program P(x,y); if x is natural then y := f(x); because the computed relation R of this program is contained in the specified relation R*. Suppose the specification is altered to: (All x Exists y) ((nat(x) & y = f(x)) AND (~nat(x) OR ~(y=f(x)))). Such a change results in an empty specified relation. In this case, the specification has been made inconsistent and there is no program that satisfies the specification.

Suppose the specification is altered to become (All x Exists y) (nat(x) & (y = f(x) OR y=g(x))) where f and g are not the same function. In this case, R* has two subrelations, R* = R1* U R2*. Effectively a new model has been added to the original R*, and there exist at least two nonequivalent programs that satisfy the specification. The first program computes the relation of the original R* (now called R1*): program P(x,y); if x is natural then y := f(x) ; and the second program: program P(x,y); if x is natural then y := g(x).

There are also many more functions that satisfy the specification: any program that agrees with f on some inputs and agrees with g on the remaining inputs will do. Clearly, it is possible to delete models without rendering the specification inconsistent. However, addition or deletion of a subrelation does not preserve the meaning of the original specification, and deleting models can invalidate an existing implementation.

Finally consider the following alteration: (All x Exists y) ((nat(x) & y = f(x)) OR (~nat(x) OR ~(y=f(x)))). This illustrates the other extreme. Such an alteration is anomalous because all programs satisfy the specification. An understanding of changing specifications requires a realization that there exist nonaxiomatic specifications in most systems, because the validity of an axiomatic specification cannot change. The question of system maintenance becomes, how is the system to be altered to conform to the changing specifications? Typically changes are made in a reactive form of software maintenance. As the validity of specifications change, the models change in one of the three fundamental ways just described. When the models change, the implemented programs may no longer be correct. Thus, systems must be adapted to satisfy the new specifications. The changing validity of the nonaxiomatic specifications is the driver of the

adaptive maintenance activity.

In practice, as the validity of nonaxiomatic specifications change, errors in processing are noticed and the users urge the software maintainers to react to the error. We believe that if the software developer can identify the valid, but nonaxiomatic specifications, it will be possible to periodically revalidate a system to see if the nonaxiomatic specifications remain valid. If not, candidate replacement specifications can be developed and validated with respect to the entire system through rapid prototyping prior to an alteration to the production system.

The need for prototyping throughout the maintenance of a production system is supported by the curve in figure 1. Here it can be seen that evolution results in changing specifications, resulting in new versions of the system. After the delivery of each version, the curve indicates the continuing problem of validation. It is clear that most systems are based on nonaxiomatic specifications resulting in the need for adaptive maintenance.

4. The Role of Prototyping

Certainly one of the greatest difficulties in developing software is the uncertainty involved in the communication with the potential user. Prototyping is known to be a way to overcome this difficulty. The CAPS system provides computer aid to address both forms of uncertainty in the domain of hard real-time system design. CAPS is targeted to assist the analyst in eliciting the requirements from the user and to validate proposed specifications. The requirements for a software system are expressed at different levels of abstraction and with different degrees of formality. The highest level requirements are usually informal and imprecise, but they are understood best by the customers. The lower levels are more precise, and are better suited for the needs of the software engineer. However, the lower levels are difficult for the customers to understand. Because of the differences in the kinds of descriptions needed by customers and developers, it is not likely that any single representation for requirements can be the "best" one for supporting the entire prototyping process. The uncertainty in specification validation manifests itself in the gap between the user-oriented specification and the designer-oriented specification.

During the process of stabilizing the requirements via prototyping, it is necessary to repeatedly move from high-level requirements to details of system behavior, and from system behavior back to high-level requirements. The prototype designers must guess the intentions of the clients based on their informal statements, and embody the client vision in a demonstrable prototype. This process is imperfect, but the demonstrated behavior of the prototype will help the customers identify differences between what they need and how the analysts interpreted their requests (i.e., errors or bugs in the analyst's interpretation will present themselves to the clients). When a bug in the system behavior is discovered, it must be traced back to the requirements to identify the specific assumptions or decisions in the requirements model that are no longer valid. After the invalid decisions have been identified and new versions have been proposed, it is necessary to trace the effects of the change back down the refinement structure to find the parts of the prototype design that are affected, so that they can be adjusted and the next approximation to the requirements can be demonstrated.

In the context of prototyping, the requirements are used as a means for bridging between the informal terms in which users and customers communicate and the formal structures comprising a prototype. We believe that a useful representation for this information is a hierarchical goal structure, where informal customer goals are refined and defined by several levels of increasingly formal and precise subgoals, with different notations used at different levels. We expect natural language to be used at the highest levels, and the prototyping language to be used at the most detailed levels, with mixtures and possibly several additional notations appearing in the intermediate levels.

The subgoals of a goal in the hierarchy are proposed interpretations for the informal parent goals. We adopt the convention that a parent goal is met whenever all of its subgoals are met. The layers of the subgoal structure correspond to decisions about proposed system behavior and how it can be packaged and presented to users. The most specific subgoals at the leaf nodes of the hierarchy are tied directly to elements of the prototype design.

Once the final requirement adjustments have been made (i.e., the requirements are validated), an effort to identify the axiomatic and nonaxiomatic specifications should be undertaken. In the original validation of the specifications, the intent is to reduce the amount of perfective maintenance. In separating the axiomatic from the nonaxiomatic specifications, we can focus the analysis supporting maintenance on the aspects of the system that are most likely to change. In the proactive adaptive maintenance, subsequent to implementation, verification, and delivery of the production software, the pro

cess of validation is repeated periodically. The non-axiomatic specifications become the focal point of revalidation since it is the validity of these specifications which changes over time. If it is discovered that a specification is no longer valid, candidate replacement specifications are produced. The candidates are validated with respect to the entire set of system specifications via the prototype. It is only after a change is validated that the production software is actually modified. A requirements tracing feature in the prototyping environment can be utilized to assist the maintainer in identifying the units of code that must be adapted in order to render the system correct. This provides a much more proactive and informed approach to change control and adaptive maintenance.

5. The Computer-Aided Prototyping System

The classical goal of rapid prototyping has been to build a model of the intended system quickly to cut down on the iterative, costly nature of software development. When the requirements cannot be completely determined or there are questions about the proposed systems, prototyping allows a model to be constructed and tested in the early stage of the development work. After testing the prototype, modifications can be made to amend the original design and a new, improved specification can be assembled for use in the coding phase. In other words, testing and refinement of requirements are done before the actual product is engineered.

In addition to the classical goal of prototyping, we pose a new goal: to establish features and methods to facilitate proactive adaptive maintenance through periodic revalidation and prototyping of the specifications of a delivered software product. Since a system is typically comprised of nonaxiomatic specifications, there are discrete intervals of time when different versions of a specification are valid. For example, if the validity of nonaxiomatic specifications of a given system change n times during the life of the associated software product, then there is a sequence of specifications which are valid during the life of the product: S1, S2,...,Sn. As the life of the product proceeds from interval Si to Si+1, the system must be adapted because it does not satisfy Si+1 - at best, it satisfies Si. Adaptive maintenance, therefore, should be viewed as building a new system based upon the new specification. The main difference between the prototyping in adaptive maintenance and the building of the original system is that a previous version exists, and the parts based on the axiomatic specifications can be reused. The CAPS system [Luqi-Ketabchi-88] is designed and constructed to increase the degree of automation in prototype development within the domain of hard real-time system development. It uses the executable Prototype System Description Language (PSDL) [Luqi-Berzins - Yeh-88] and consists of an integrated set of software tools including user interface, syntax-directed editor, graphic editor, execution support system, design database, software base, and design management system.

5.1. Prototype System Description Language

PSDL provides a simple representation of system decompositions using data flow diagrams augmented with non-procedural control constraints and timing constraints (maximum response times, maximum execution times, minimum inter-stimulus periods, periods, and deadlines). The language models both periodic and data-driven tasks, and both discrete (transaction-oriented) and continuous (sampled) data streams. The CAPS system provides automated tools for generating static schedules to guarantee hard real-time constraints as well as an execution support system that generates code for adapting, interconnecting, and controlling the execution of reusable software components. Through PSDL, the CAPS system provides the tools necessary to model real-time systems for the purpose of requirements validation. CAPS is based upon PSDL and provides an environment for automating prototype construction and maintenance.

5.2. User Interface

The graphic editor, in the User Interface, is a tool which permits the user/software engineer to construct a prototype for the intended system using graphical objects to represent the system [Linton, TAE]. These objects include operators, inputs, outputs, data flows, and operator loops. The syntax directed editor is used by the user/software engineer to enter additional annotations to the graphics. A browser allows the analyst to view reusable components in the software base. An expert system provides the capability to generate English text descriptions of PSDL specifications to facilitate common understanding of PSDL components by users and software engineers alike, thereby reducing design errors and improving the ability to validate or revalidate specifications.

5.3. Software Database System

The software database system provides the designer access to reusable software components for realizing given functional (PSDL) specifications, and

consists of a design database, software base, and software design management system.

The design database [Borison, Nestor] contains PSDL prototype descriptions for all software projects developed using CAPS. The software base contains PSDL descriptions and implementations for all reusable software components developed using CAPS. Prototyping with the software base speeds up evolution by providing many different versions of commonly used components [Steigerwald], making it easier to try out alternative designs. The software design management system manages and retrieves the versions, refinements and alternatives of the prototypes in the design database and the reusable components in the software base.

5.4. Execution Support System

The execution support system (ESS) provides the user with views of the dynamics of a specification. With the ESS, the analyst can demonstrate the prototype for the purpose of requirements validation. The ESS consists of a translator, a static scheduler, a dynamic scheduler, and a debugger. The translator generates code that binds together the reusable components extracted from the software base. The static scheduler allocates time slots for operators with real time constraints [Mok] before execution begins. As execution proceeds, the dynamic scheduler invokes operators without real-time constraints in the time slots not used by operators with real-time constraints. The debugger allows the designer to interact with the execution support system.

6. Domain Specificity and Requirement Traceability

Using CAPS to engineer requirements offers clear advantages over determining requirements manually. The prototype system description language PSDL is focused on the domain of hard real-time systems and as such offers a common baseline from which users and software engineers describe requirements. Defining requirements in a domain specific language is more efficient and and results in fewer errors because it constrains the way users describe a particular requirement. Also, the interpretation of requirements stated in a language like PSDL are unambiguous, whereas requirements stated in English are often misunderstood. Thus, as ambiguity is reduced through CAPS, the requirements validation becomes more certain.

In most software engineering efforts requirements are volatile, changing often over the course of software development and maintenance. Requirements traceability is essential to accurately map changed requirements into the prototype or the implementation. CAPS offers basic requirements traceability through the "by requirements" statements. This method allows engineers using the design database to readily locate the portions of code that implement a particular requirement and make the appropriate changes. The requirements tracing feature is an example of a CAPS feature which will assist the software maintainer in tracing a nonaxiomatic specification, the validity of which has changed, to its associated program unit.

Figure 2 illustrates the prototyping process which can fit into the requirement, design, implementation, and maintenance phases of other software process models. The four major stages in the CAPS process, prototype design, construction, execution, and debugging/modification, support an iterative prototyping life cycle [Luqi-Ketabchi-88]. The initial prototype design starts with an analysis of the problem and a decision as to which parts of the proposed system are to be prototyped. Requirements are then generated, either in English or some formal notation. These requirements may be refined by asking the user to verify their completeness and correctness. After the preliminary design is completed, construction of the prototype may begin. When supplied with the design, CAPS will guide the user to produce a PSDL prototype representing the specification and the design of the intended system. This prototype is then fed to the execution support system which translates the PSDL specification into Ada code and evaluates its behavior. Lastly, debugging and modification are done over the entire CAPS utilizing all the tools. The purpose of CAPS is not to design a system, but rather to test and validate the specification.

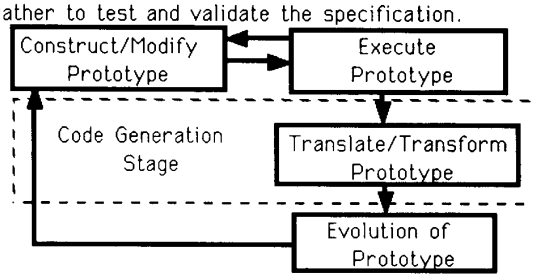

Fig. 2 Rapid Prototyping Process

The user interface of CAPS is responsible for the sequence control throughout prototype development and for the assurance of continuity of the various levels of refinement during prototype construction. This interface possesses the knowledge of the functions of all the components within CAPS and is able to interpret what the user is doing at any time and to generate queries to find out what the

user wants whenever the system is not sure of the user's intention [Raum-88].

The code generation stage focuses on transforming and augmenting the prototype to generate the production code. Prototypes are built to gain information to guide analysis and design, and support automatic generation of the production code. Once the code is produced, CAPS can be used to periodically revalidate the specification to facilitate and control adaptive change to delivered software.

7. Conclusions and Summary

We feel that the prototyping framework further suggests the software process model which follows: (1). Divide a system into its axiomatic and nonaxiomatic specifications. (2). Validate the original specifications via rapid prototyp ing. This step is iterative based upon effective interaction with the client. (3). Develop production system based upon final prototype. (4). Verify production system according to previously validated specifications. (5). Maintain the software in a proactive rather then reactive manner.

In step 5, we propose that maintainers of a software system periodically review all specifications to determine those which are axiomatic and those which are nonaxiomatic (because of the possibility of misclassifications) and revalidate the nonaxiomatic specifications since these are the specifications which may become invalid over the life of the project. Prior to changing the production software, the specified change is validated with the entire system of specifications to ascertain side effects of the change. This is to be accomplished via the prototype. Thus the effect of change becomes more predictable and manageable. Once the change has been validated through the prototype, the requirements tracing feature can be used to identify the program unit(s) which need(s) to be adapted.

CAPS suggests a form of maintenance which is more scientific, proactive, and focused. Although CAPS is targeted at real-time systems, the general philosophy of the CAPS framework should facilitate the management of uncertainty in any environment.

References

[Boehm] B. Boehm, "A Spiral Model of Software Development and Enhancement," Computer, Vol. 21, No. 5, May, 1988, pp. 61-72.

[Borison] Borison, E., "Program Changes and Cost of Selective Recompilation", Technical Report CMU-CS-89-205, Computer Science Department, Carnegie-Mellon University, July 1989.

[Dun] Dunn, R. Software Quality Concepts and Plans, Prentice-Hall, Englewood Cliffs, N.J., 1990, pp. 142-145.

[Leh] M.M. Lehman, "Programs, Life Cycles, and Laws of Software Evolution," Proceedings of IEEE, Vol. 68, No. 9, September, 1980, pp.1060-1075.

[Linton] Linton, M. A., Vlissides, J. M., and Calder P.R.,"Composing User Interfaces with InterViews", IEEE Computer, February 1989.

[Luqi88a] Luqi, and Ketabchi, M.,"A Computer-Aided Prototyping System", IEEE Transactions on Software Engineering, October 1988.

[Luqi88b] Luqi, Berzins,V., and Yeh,R., "A Prototyping Language for Real-Time Software", IEEE Trans. on Software Engineering, October 1988.

[Mills]Mills,H. "Software Development," IEEE Trans. on Software Eng. SE-2, 4 (Dec, 1976), 265-273.

[Mok] Mok, A., "A Graph Based Computational Model for Real-Time Systems", Proceedings of the IEEE International Conference on Parallel Processing, Pennsylvania State University, 1985.

[Nestor] Nestor, J., "Toward a Persistent Object Base", in Advanced Programming Environments, vol. 244, Lecture Notes in Computer Science, Springer-Verlag, 1986, p.372-394.

[Ram] C.V. Ramamoorthy and Daniel E. Cooke, "The Correspondence Between Methods of Artificial Intelligence and the Production and Maintenance of Evolutionary Software," Proceedings of the 3rd International IEEE Conference on Tools for Artificial Intelligence, November, 1991, pp. 114-118.

[Raum] H. Raum, "Design and Implementation of an Expert User Interface for the Computer Aided Prototyping System," Master's Thesis, Computer Science, Naval Postgraduate School, Monterey, Calif. Dec. 1988.

[Steigerwald] Steigerwald, R., Luqi, and McDowell, J., "Rapid Prototyping with Reusable Software Components: Methodologies for Component Storage and Retrieval", submitted to Journal of Software Engineering and Knowledge Engineering for publication.

[TAE]Transportable Applications Environment (TAE) Plus, National Aeronautics and Space Administration, Goddard Space Flight Center, January 1990.

Acknowledgement

Research sponsored by the National Science Foundation under grant numbers CCR-9058453 and CDA-9015006; and by the Air Force Office of Scientific Research (AFSC), under contract F49620-89-C-0074.