1989

Multi-Level Software Analysis and Testing in Evolutionary Software Development

Luqi

Naval Postgraduate School

Multi-Level Software Analysis and Testing in Evolutionary Software Development

Luqi

Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

ABSTRACT

This paper presents a view of research directions in multi-level software analysis and testing. Software analysis is needed at different levels of the development process. Appropriate research goals are identified for exploring software analysis at each of these levels in order to produce good quality software at reduced cost.

1. Introduction

The goals of software analysis and testing are to measure essential software properties and to enable development of software systems with specified ranges for those properties. Reliability of software products is gaining increasing importance, particularly for systems whose malfunction may result in loss of human life, compromise of national security, or massive loss of property. Software properties related to reliability include constraints on functional behavior, timing, and storage space. Software maintenance is also a major concern because it typically accounts for more than half the cost of a software system. Software properties related to maintenance include the subset of a software system affected by a proposed change, the effects of a software modification on a given reliability property, and invariant relationships among members of a family of software systems.

Classical approaches to reliability are based on the assumption of an imperfect software development process, and hence focus on detecting errors and measuring reliability properties of software products. More recently proposed approaches explore special types of computer-aided software development processes which can guarantee the resulting software products are free of particular classes of errors via properties of the formalized development process. Research on software analysis and testing should address processes for achieving desired properties of software products as well as methods for measuring those properties.

2. Levels of Software Analysis and Testing

Different types of software analysis and testing are appropriate at different stages of software development, as summarized in Fig. 1.

2.1. Requirements Level

The requirements level established the goals for a proposed system and formulates models of the problem and the expected environment of the proposed system.

Level	Type of Analysis and Testing
Requirements	consistency: truth maintenance model validation: simulation and proof subgoal verification: prototyping and proof
Specification	adequacy: prototyping, operational scenarios consistency: type and domain checking safety: proofs validation: paraphrasing, views, simplification
Design	verification: proof of decomposition liveness: deadlock and starvation checking robustness: impact of degraded hardware design for testing: control and observation performance: complexity analysis feasibility: satisfiability proofs
Coding	synthesis: meaning-preserving transformations performance: time and space analysis liveness: proof of (clean) termination real-time: analysis of scheduling methods generic units: analysis of component families error detection: complete test sets error location: weakest preconditions
Evolution	change impact: symbolic differences restructuring: meaning-preserving transformations

Fig. 1 Types of Software Analysis and Testing

An important aspect of requirements analysis is achieving and maintaining consistency as the analysts discover and record the requirements. A promising approach to this problem is providing automated support for calculating and maintaining derived properties and consequences of the requirements, and for tracing dependencies to determine the causes of conflicts and inconsistencies. Better algorithms for this process and primitives suitable for expressing and effectively maintaining dependencies in software requirements should be investigated.

Another aspect of requirements analysis is modeling the environment of a proposed system. Especially for embedded software systems, an accurate formal characterization of the system to be controlled is essential for assessing the effectiveness of the control software. The environment of such a system must often be simulated or otherwise

formally analyzed to enable safe and meaningful testing or analysis of the embedded software system. Systematic methods for validating and testing the formal models of the environment against the properties of the actual physical systems they represent are needed. Both analytical and experimental methods should be explored to establish that the formal environment models used in other software analysis and testing activities are adequate representations of reality.

Many critical software systems are embedded systems, which means that the software is part of a larger system. Thus an essential part of checking the adequacy of the requirements for the software is checking that any system meeting the requirements will be sufficient to meet the requirements for the larger system in its intended operational context. Hard real-time constraints in an embedded system are often motivated by the requirement to control the larger system to ensure it remains within a given range of operating states. For example, the cycle rate of an auto-pilot must be sufficiently high to ensure that the airplane remains within a given radius of its planned position at all times. At the current time, lower level requirements are usually formulated based on past experience and informal guidelines rather than on systematic derivations or verification procedures with respect to the higher level requirements. Both formal and experimental methods for systematically establishing such properties are needed. Required supporting technology for this process includes computer-aided construction of prototypes.

2.2. Specification Level

The specification level is concerned with defining the interface of a proposed system, both at the functional and the command representation levels.

The primary measure of the adequacy of a specified interface is whether it will meet the needs of the user. This question is best addressed by experimental rather than analytical techniques because it addresses the problem of checking the correspondence between a formalized specification and the actual and informal needs of the users. One way of approaching this problem is via prototyping and operational scenarios. Operational scenarios are common tasks in the customer's problem domain, expressed in the user's terms. Such scenarios serve as test cases for the specifications, whose purpose is to determine whether a proposed interface is adequate for carrying out all of the tasks the users will have to perform. Such a test passes if the facilities provided by the proposed system interface can be combined to carry out the tasks in the operational scenario, and provide a systematic means for exercising a prototype in a demonstration to the users. Systematic methods for deriving sets of scenarios from a requirements document, coverage criteria, and experimental evaluation of the effects of such coverage criteria on change requests to the affected interfaces during system maintenance should be investigated.

A related concern is validating a formal specification, to ensure that it correctly captures the intentions of the users. While this is an informal process, it can be aided by formalized and automatable procedures. Some of the processes involved are paraphrasing, projection, and simplification. Paraphrasing is the process of transforming a formal specification into a form that a user can understand, while preserving its meaning. Projection is the process of extracting the parts of a specification relevant to a particular user or task, while hiding other details. Simplification is the process of transforming a formal specification into a simpler form with an equivalent meaning. These three processes can

be combined to help users selectively review formal specifications using representations they can understand. The research questions in this area concern certifying the transformations to ensure they preserve the meaning of the specification and experimentally evaluating the effectiveness of different representations for communicating with untrained users.

Consistency of a specification is another common concern, especially for large and complex systems. Since consistency is a property of a formal document, it can be addressed by analytical techniques. Some aspects of consistency checking that need further development are type and domain consistency checking. Type checking at the specification level is more difficult than the corresponding problem at the code level because types can have subtypes defined by semantic considerations. Domain checking is the process of ensuring that partial functions or predicates are used only within their domain of definition, and that partially defined generic units are instantiated only with actual parameters in their respective domains of definition. Logical inference capabilities are necessary for both of these kinds of specification analysis.

Another concern with formal specifications is checking safety properties. For example, past research projects have been concerned with whether a proposed operating systems kernel satisfied certain security properties, such as the impossibility of transmitting classified information from a process with a high security classification to an unauthorized process. The goals of safety analysis procedures are to identify cases where the specifications allow behaviors violating the safety properties, or to certify that no such cases exist. Systematic procedures for this process are needed because the connection between a formal specification and a safety property can be quite indirect and can require extensive reasoning and analysis to establish.

2.3. Design Level

The design level is concerned with the decomposition of a problem into a hierarchical structure of independent modules. Such a decomposition consists of interconnection information and formal specifications for the components.

The primary reliability property of a decomposition structure is whether it will correctly realize the specification at the next higher level. This problem is subject to mathematical proof techniques. The problem is easier to solve than the general proof of correctness problem at the code level because the module interconnection language is can be considerably simpler than a programming language. Most of the analysis can be carried out at the specification level, since the problem is to check whether a given combination of specified components will satisfy the required properties of the composite. Research questions in this area involve the best choice of interconnection primitives to support effective and efficient inference procedures.

Another type of property of interest for parallel and distributed systems is liveness. Techniques for checking for potential deadlock or starvation conditions in such a design are desired. Such techniques can be based on the combination of fast graph algorithms with satisfiability checking for paths leading to potential problems. The main research questions are finding efficient special purpose analysis techniques that can address semantic issues which are neglected by classical Petri net techniques or involve infinite graphs if encoded as standard Petri nets.

An important class of analysis involves the effects of degraded hardware on the properties of a design, relative to a mapping of software components to hardware components. This kind of analysis is essential for achieving reliable fault tolerant systems, especially those with distributed implementations. In addition to certifying that proposed configurations realize given degrees of fault tolerance, automatic derivation of the implied constraints on allocation of software functions to hardware units is desirable.

Automated assessment or augmentation of a design for supporting testing is another issue at the design level. Facilities for control and observation of closed modules such as abstract data types and machines are needed to support testing. Guidelines for what attributes of a system need to be controlled or monitored for effective testing must be developed, along with automated techniques for generating the code that realizes the control and monitoring functions to be added during testing. Such an investigation should be coupled with an analysis of the impact of the additional code on time and space requirements, and techniques for automatically compensating for their effects in checking timing and space constraints.

Evaluation of a design for time and space performance is another kind of software analysis that has potential importance. Automated support for classical complexity analysis is needed, along with estimates for the ranges of input sizes and constant factors determined by classes of algorithms.

A final consideration is satisfiability. The satisfiability of a specification can be established if an implementation can be produced and certified to be correct. However, it would be useful to determine whether it is possible to satisfy a given specification before the implementation is attempted, and in cases where it is not, to characterize the set of inputs for which the specification is impossible to meet. Analytical techniques for constructing weakest infeasible preconditions characterizing this set of inputs should be explored.

2.4. Code Level

The best way to achieve quality is to systematically prevent errors. Automatable methods for synthesis of efficient code from formal specifications via meaning-preserving transformations should be investigated. Of particular interest are systems that can choose transformations without explicit human guidance, or with guidance from general declarative advice that can be formulated without explicit reference to the details of the current state of the derivation and does not require explicit human interaction during the derivation process.

Accurate performance analysis requires detailed code and knowledge about properties of a particular compiler and target machine. Generic table driven methods for performing such analysis, and for relating design-level properties of abstract algorithms to detailed properties of actual machine-level implementations and compiler optimizations is needed to accurately certify correctness of programs with hard real-time and real-space constraints. Research problems in this area include formal modeling of implementation-specific properties and constraints in ways that can be combined with implementation-independent analyses of abstract programs.

Another problem is certification of clean termination. This problem gains new dimensions in parallel and distributed systems, where termination can be influenced by

scheduling properties and hardware failures. Research questions include models and techniques for analyzing programs in these domains.

Analysis of real-time systems includes analysis of scheduling methods to determine whether a proposed scheduling discipline will meet specified deadlines under all possible operating conditions. Research questions in this area have flexible scheduling methods, the effects of shared resources, overload resolution policies, and remote communications as major concerns.

The problem of certifying generic code units or families of related programs generated by meta-programming schemes is a major concern in systems for managing reusable software. A software component is most effectively re-used if it is flexible and can be adapted to many needs. Such a component often corresponds to a family of related program units with an unbounded number of elements. The problems of testing and analyzing the reliability of such program families is an important research question.

Classical testing approaches need more foundational work on the construction of complete test sets. A complete test set is a set of test cases which is guaranteed to detect any error in a particular well-defined class of errors. More work is needed on the construction of finite complete test sets, an on characterizing the set of faults whose absence is guaranteed by successful execution of the test set. Such work should include automated techniques for constructing the required test oracles from the formal specifications of the code to be tested.

An aspect of code analysis of great practical importance is error location. One approach to this problem is to derive weakest preconditions for suspected pieces of code, to characterize the space of inputs for which the code fails.

2.5. Evolution Aspects

Software maintenance is acknowledged to be more difficult and error prone than the initial development. An important kind of software analysis for this part of software development is characterization of the effects of a change to a software system. Symbolic representations for the parts of the input space and the output space of a program affected by a given change to the code are useful for testing and evaluating a modification for conformance with the expected results. Computer-aided identification of the parts of a specification affected by a given requirements change, the parts of a design affected by a given specification change, and the parts of the code affected by a given design change are also important areas for research.

When changing a software system, it is often necessary to reverse an earlier design decision while preserving the later ones. Automated construction and application of meaning-preserving transformations that accomplish this is an important research problem.

3. Conclusion

Advances in software analysis and testing are essential for realizing trusted software systems. Work in this area should be expanded beyond the traditional domain of testing code in a programming language to include software products at all stages of development, from requirements analysis to system evolution. Further work on testing at the code level is also needed, to enable firm conclusions to be drawn from finite sets of test

cases constructed by definite and effective methods. Software analysis techniques addressing properties of parallel, distributed, real-time, and knowledge-based systems should be explored as well as those for sequential systems. Well founded and mechanizable analysis techniques are needed for meta-programming and program transformation systems as well as for individual software products.

DISTRIBUTION LIST

(1)	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
(2)	Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93943	2
(3)	Center for Naval Analysis 4401 Ford Avenue Alexandria, VA 22302-0268	1
(4)	Director of Research Administration Attn: Prof. Howard Code 012 Naval Postgraduate School Monterey, CA 93943	1
(5)	Chairman, Code 52 Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	1
(6)	Chief of Naval Research 800 N. Quincy Street Arlington, Virginia 22217	1
(7)	Naval Postgraduate School Code 52Lq Computer Science Department Monterey, CA 93943	100