Dagstuhl Seminar on Practical Methods for Code Documentation and Inspection

Egon Börger (Italy), Dave Parnas (Canada), Paul Joannou (Canada)

October 11, 1997

The aim of the workshop was to bring together software engineering researchers from accademia and software engineers from industry to discuss the state of the art of methods for code documentation and code inspection with focus on their industrial strength and on their relevance for safety critical software certification.

The work was done through seminar talks (see the abstracts below), evening working groups and a code inspection session. The discussions were focussed on the properties a documentation and certification method must satisfy to be appropriate for large scale use, on the evaluation of current methods under the aspect of their industrial strength and on future directions for research in this area.

A particular aspect which has been discussed extensively is the role of formal methods for code verification, including the requirement analysis and specification as part of the design process. A related aspect wich has been investigated is the relation between verification and formally supported validation techniques and in particular the relation between mechanical (tool based) techniques and creative aspects of code inspection.

Here are some of the reccomendations which grew out of the discussions. "Active" reviews were recommended. It was recommended that interfaces describe the behavior that can be expected with normal input and with exception cases (noting that nothing can be guaranteed when assumptions are violated). It was recommended to use techniques of paraphrasing the code by the reader and having the reviewers fill out questionaires. These questionnaires prove that the documents have actually been read and are useful for finding the required information. Another interesting idea is to assign the reviewers to look for different specific faults in the document or code based on their skillset and to strip out the comments for the review. Hypertext tools could offer great traceability from requirements to design specs and to the code and test cases.

Below we list the abstracts of the talks which have been delivered during the seminar. We thank Schloiss Dagstuhl for offering hospitality. Thanks also to Luca Mearelli for his help in compiling this report.

Egon Börger Dave Parnas Paul Joannou

Integrating ASMs into the Software Development Life Cycle

Egon Börger
Università di Pisa, Italy
boerger@di.unipi.it
Luca Mearelli
Università di Pisa, Italy
luca@tex.odd.it

We show how to integrate the use of Gurevich's Abstract State Machines (ASMs) into a complete software development life cycle. We present a structured software engineering method which allows the software engineer to control efficiently the modular development and the maintenance of well documented, formally inspectable and smoothly modifiable code out of rigorous ASM models for requirement specifications. We show that the code properties of interest (like correctness, safety, liveness and performance conditions) can be proved at high levels of abstraction by traditional and reusable mathematical arguments which where needed are amenable to computer verification. We also show that the proposed method is appropriate for dealing in a rigorous but transparent manner with hardware-software co- design aspects of system development. The approach is illustrated by developing a C ++ program for the production cell control problem by Lewerentz and Lindner (see Springer LNCS 891). The program has been validated by extensive eperimentation with the FZI production cell simulator in Karlsruhe.

The paper is published in the special ASM issue of the Journal of Universal Computer Science (see http://www.iicm.edu/jucs_4).

Reverse engineering of Fortran code using algebraic specifications

Sophie Cherki
University of Paris-South, France
Christine Choppy (giving the talk)
University of Nantes, France

This talk describes an experience in trying to help maintaining existing Fortran 77 code of a large industrial application. Our approach is to start with reverse engineering this code using algebraic specifications to provide an abstract description of its functionalities. This implies an active reading of the code (together with the comments in the code) which lead us to find out interesting bugs and anomalies.

The resulting algebraic specification consists in a graph of specification modules; the axioms may be first order formulae, but we mainly used conditional equations. The code of this application is structured in such a way that, most of the time, a module implements a few functionalities. The structure of the algebraic specification reflects this code structure.

The key issues are to find out the signatures and the axioms for the specification modules. Fortran 77 exhibits only predefined types and anonymous array types. As a consequence, extracting the signature of a specification associated to a Fortran module raises various difficulties and we present how to overcome them. Extracting the specification axioms is achieved by means of identifying unit actions within the code, composing their associated equations, and simplifying the - rather unreadable - resulting expression in order to obtain axioms that are easier to read (this last step requires the use of theorem proving).

The resulting specification may serve both as a precise documentation, and as a basis for the development of the new version of the code, where both useless code, anomalies and bugs are removed.

Specification of an Adaptive Cruise Control with Z and Statemate

Heiko Doerr

Adaptive cruise control systems support car drivers by controlling the vehicle speed adaptively to a vehicle in front. They are typical examples for software-based, safety-related, embedded systems in automotive electronics.

In the talk, parts of the requirements specification of such system are presented. Different aspects - basis, functional and safety requirements - are separately specified by a methodically supported combination of Z, statecharts and data flow diagrams. The specification method is part of an overall software technology for the development of complex, safety-critical embedded systems. It provides a solid base for the later phases of the software development process especially for testing.

Abstract State Machine (ASM) Specification of Embedded Control Systems: Documentation and Validation

Uwe Glæsser

We use the steam boiler control specification problem [1] to illustrate how the ASM approach to modeling and validation of computer based systems can be exploited for a systematic and well documented development of formally inspectable code.

Starting from an informal problem description we derive a C++ implementation of the control program through a hierarchy of stepwise refined abstract machine models. These mathematical models document the design decisions that lead from the requirement specification, which is formally expressed in terms of our ASM ground model, to the final implementation, a sufficiently refined ASM, which is directly encoded in C++.

For each of these models we can prove a number of properties under precisely stated assumptions about the behavior of the physical environment.

[1] J.-R. Abrial, E. Börger, and H. Langmaack, editors, *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, volume 1165 of *LNCS (State–of–the–Art Survey)*, Springer-Verlag, 1996.

Software Inspection and Module Interface Specification

Dan Hoffman
University of Victoria

Software inspection is widely practiced in industry today. Numerous studies provide strong evidence that inspection is effective in the sense that the the technique is teachable, the results are repeatable, and the benefits significantly outweigh the costs. As practiced today, inspection is based on verbal paraphrasing of source code. Usually there is little or no documentation.

Over the past 10 years, we have developed "focused inspections", which improve the inspection process by using:

- documentation specially designed for inspection,
- classical program proof strategies adapted to the inspection context, and
- inspections focused on specific faults, such as pointer errors.

We have found that module interfaces are an excellent target for focused inspections.

Focusedinspections are described in detail in the text

> *Software Design, Automated Testing, and Maintenance*,
> D. Hoffman and P. Strooper,
> Interntional Thomson Computer Press, 1995.

An industrial experiment in focused inspection is presented in

> *Inspecting Module Interface Specifications*,
> A. Jackson and D. Hoffman,
> Journal of Software Testing, Verification, and Reliability, Vol. 4, 1994.

Ontario Hydro Approach to Safety Critical Software Engineering

Paul Joannou
Ontario Hydro
paul.k.joannou@hydro.on.ca

Ontario Hydro and Atomic Energy Canada Limited (AECL) have developed an approach to engineering safety critical software for use in nuclear power plants. The approach is documented in a family of standards, procedures and guidelines that are used by both companies. The approach consists of the following:

- a guideline for categorization of software to determine its criticality relative to nuclear safety
- high level standards that specify the methodology independent requirements to be met by the software to provide the necessary level of confidence in the software that is commensurate with its criticality
- detailed procedures that document the specific methods to be used to carry out each step in the software engineering process. Procedures may be common to multiple categories or may be specific to one category
- a guideline for qualification of predeveloped software, such as commercial, off-the-shelf packages, or embedded firmware.

Safety critical software is the most critical, and hence requires that the most rigorous methods be applied to gain a very high level of confidence in the software. The standard for software engineering of safety critical software requires that a process of stepwise refinement be used to develop the software, where the requirements for the software are first documented and verified, the software design is documented and verified and finally the code is produced and verified.

The standard requires that mathematically precise specifications of requirements and design be produced in a systematic manner that minimizes the probability of introduction of error in the forward going engineering processes. It also requires that rigorous methods be applied for the verification of the software to maximize the probability of detection of errors in the software. Verification processes include reviews, mathematical based verification, phased testing, software hazards analysis and statistically valid random testing.

This approach to engineering safety critical software has been applied to three completed projects, has been revised to reflect experience gained, and is currently being applied to 6 different shutdown systems under development. Experience to date has shown that the application of these techniques results in early detection and removal of errors, and that the costs for the techniques are fairly well balanced among the verification processes, although the mathematical verification processes still have scope of improvement.

Categorization

Software applications are categorized using a risk based approach that takes into account both the consequence of software failure and the reliability requirement allocated to the software. The categorization process has been found to be a useful step in assessing a proposed system design since it sometimes finds that undue reliance has been placed on a software based subsystem, and that with a small system design change the reliance on the software can be reduced without undue impact on the system design.

Software Requirements Specification (SRS)

A good requirements specification should have the attributes of completeness, correctness, consistency, verifiability, modifiability, traceability, and understandability. We have found that by using an approach of specifying a mathematical model that defines the required behaviour of the system and then specifying allowable tolerances around this ideal behaviour that the implementation must fall within, has provided an effective means of achieving many of these attributes. We use a discrete, finite state machine model to specify the ideal behaviour. Tabular representations of the nextstate and output functions that characterize a FSM are used and have been found to be effective. The tables are easily understood by a wide audience, are well suited to the piecewise continuous functions typically implemented in computer systems and do not unintentionally constrain the implementation.

Software Design Description (SDD)

The software design description describes the design to a sufficient level of detail so that no further refinement of the module structure, module interfaces, data structures, or databases is required in the code. A good software design should have the attributes of high cohesion, loose coupling, low complexity, extensibility, reusability, portability and stratification.

Information hiding is the fundamental design technique used to achieve the above attributes. Information hiding requires that software modules be defined based on a desire to encapsulate areas of the software that are likely to change so that future changes can be made without undue impact on multiple modules. The required behaviour of the programs within a module are defined in the design description by program function tables that define the results of executing a program in terms of the inputs to the program.

Reviews

Reviews are used to identify ambiguities and incompleteness, to check documents for conformance to preceding documents and to check for conformance to procedures. The methods used for review include active reviews, structured walkthroughs and inspections guided by checklists.

Systematic Verification

Systematic verification of the software design description is used to verify, using mathematical verification techniques or rigorous arguments, that for every output, the behaviour for that output, as defined in the SDD, is in compliance with the requirements for the behaviour imposed by the SRS, and to identify any functions outside the scope of the requirements specified in the SRS and to check that justification has been provided for their existence.

Systematic verification of the code is used to verify, using mathematical verification techniques or rigorous arguments, that the behaviour of outputs with respect to inputs is the same as that specified by the SDD for the entire domain of the inputs.

Testing

Testing is conducted in three phases, each targeting a different class of errors to uncover and each with its own set of input documents upon which it is based.

Unit testing is done of individual programs and is based on white box test coverage criteria such as statement coverage, branch coverage, decision coverage and path coverage criteria. The unit testing test cases are based upon analysis of both the code and the software design description.

Integration testing is done to test that the software modules integrated together and with the target hardware and pre-developed software meet the requirements specified in the SRS, to find errors in the software, hardware, and pre-developed software interfaces, and to find errors in handling stress conditions, timing, fail-safe features, error conditions, and error recovery.

Validation testing is done to test that the entire executable code integrated with the target hardware and any pre-developed software meets the requirements specified in the system level requirements and design documentation.

Software Hazards Analysis

The objective of the Software Hazards Analysis is to verify that the software required to handle system failure modes does so effectively, to undertake a review of the code from the safety perspective (as orthogonal to the functional perspective) and thus identify any failure modes that can lead to an unsafe state and make recommendations for changes, to determine sequences of inputs which could lead to the software causing an unsafe state and to make recommendations for changes.

Hazards analyses are performed at several points in the software design process. The objective is to explicitly address the safety reliability view of the design (as orthogonal to the functional view of the design).

Reliability Qualification

The objective of Reliability Qualification is to demonstrate that the reliability hypothesis is achieved for the executable code (integrated with the target hardware and any pre-developed software) with the degree of confidence necessary to meet the reliability requirements.

Experience to Date

Three projects have been completed using the above approach. Experience from these projects indicates that the techniques are effective at minimizing introduction of errors and effective at maximizing the detection and removal of errors in the software. Most errors are detected and removed before code is produced. This has resulted in very few errors being detected by testing, despite very comprehensive testing being conducted.

The costs of the approach is considered balanced with no one process taking a disproportionate amount of time. To date the systematic verification process has been done without tool support and so it should be possible to reduce the effort once mature tools are available.

Future Directions

Experience gained in applying these techniques is being used to revise the standards, procedures and guidelines that capture the technology. Tool support is being developed. One challenge for the future is to scale the technology to be applicable to some of the category two systems that are more complex than the very simple, safety critical systems used in nuclear power plants. Another challenge is to provide educational infrastructure so that future designers may be educated in these techniques.

Compiler Implementation Verification as a Code Documentation and Inspection Problem

Hans Langmaack
Institut für Informatik und Praktische Mathematik der
Christian-Albrechts-Universität zu Kiel
hl@informatik.uni-kiel.de

After 40 years of practice and theory in compiler construction and 30 years of experience and teaching in software engineering we still observe that certification institutions are certifying safety critical high level languages programs only in combination with their generated machine programs. Certification institutions do not trust any compilers. And the institutions are very right: Whereas thought errors detected in processor hardware are felt like sensations, thought errors in software, even in systems software, are commonplace.

It is high time to reverse this trend. Informaticians should concentrate their abilities, experiences and insights for safe mastery of realistic systems software. Realistic compilers for realistic programming languages on hardware processors are especially addressed. Correct compilers play a central role in construction of trustworthy application and systems programs. We need trusted development environments for application programmers and system software engineers such that they can concentrate in software specification and high level implementation and need not spend their time with compilation problems and machine code inspection again and again.

Constructing fully correct realistic compilers useful for safety critical applications must master both compiling specification verification and compiler implementation verification. The first problem is to specify and prove semantically correct a mathematical translation function from high level source programs to realistic target machine code. The second problem is to correctly refine and implement such function in a host language for which there is a running and trusted host compiler or down in machine language of a real host processor. Clear, initially only the second way is trustworthy.

Literature on compiler verification is dealing almost only with the first problem. The DFG-project "Verifizierte Übersetzer - Verifix" together with Univ. Karlsruhe (G. Goos) and Univ. Ulm (F.W. von Henke) attacks full realistic compiler verification, especially the second problem. The problem is realistically managable by a-posteriori-control (German: Beweis durch Probe) and a diagonal technique which exploits assumed hardware correctness to avoid redundant double checking especially of low level code.

Refining an ASM Specification of the Production Cell to C++ Code

Luca Mearelli
Universitá di Pisa
(luca@tex.odd.it)

We present here the transformation to C++ code of the refined ASM model for the production cell developed in the paper "Integrating ASMs into the Software Development Life Cycle" (see this volume) which serves as program documentation. This implementation is a refinement step and produces code which has been validated through extensive experimentation with the production cell simulator of FZI Karlsruhe.

The paper is published in the special ASM issue of the Journal of Universal Computer Science (see http://www.iicm.edu/jucs_4).

Development of Critical Systems: An Approach Based on Modular Logic Specifications

Angelo Morzenti

I presented the results of a project regarding the development of a system controlling the load balance among a set of generators in a pondage power plant. I illustrated the various phases of the project, namely education and training, requirements elicitation and specification, requirements validation and verification planning, design and coding. I evaluated the relative costs of the various phases comparing them with those in a previous, similar project. Based on this and other, preceeding and successive projects, I drew the following (somehow provocative) conclusions. Code and design documentation can be a by product of a process that starts from requirements specifications. Providing design and coding documentation a posteriori is much harder and costly: it could be considered as a reverse engineering activity. Similar remarks hold for the validation and verification activities, provided that the specification includes not only user requirements but also a detailed description of a strategy for solving the problem. The importance of formal specifications can be hardly overstressed, as they provide a mathematical object on which one can apply systematic or automatic procedures for validation and verification. Also very important is to modularize specification, possibly combining modularization with refinement. By these means one can gradually move from abstract (descriptive, nondeterministic, constraint-like, input/output-based...) requirements to more concrete (detailed, deterministic, operational, state-based) design specifications. After this modularization/refinement process is carried out, the mapping of the specifications to a program in some object-oriented programming language is very much facilitated. In our experience coding errors (i.e., errors deriving from misinterpretation of correct and clearly stated requirements) are rather rare, therefore we consider validation a more critical and useful activity than verification. All the above remarks hold in the case of non-standard critical applications; when the application domain is well known and studied, (e.g., the compiler design domain) a systematic or even automatic techniques are available to move from specifications (e.g., grammars, in the case of compiler design) to software code or hardware.

Software Inspections We Can Trust

David Lorge Parnas
NSERC/Bell Industrial Research Chair in Software Engineering
Communications Research Laboratory
Department of Electrical and Computer Engineering
McMaster University Hamilton, Ontario Canada L8S 4K1

Software is devilishly hard to inspect. Serious errors can hide for years. Consequently, many are hesitant to employ software in safety-critical applications and all companies are finding correcting and improving software to be an increasingly burdensome cost.

This talk describes a procedure for inspecting software that consistently finds subtle errors in software, software that is believed to be correct. The procedure is based on four key ideas:

- All software reviewers are actively using the code.
- Reviewers exploit the hierarchical structure of the code rather than proceeding sequentially through the code.
- Reviewers focus on small sections of code, producing precise summaries that are used then inspecting other such sections.
- Reviewers proceed systematically so that no case, and no section of the program, gets overlooked.

During the procedure, the inspectors produce and review mathematical documentation. The mathematics allows them to check for complete coverage; the notation allows the work to proceed in small systematic steps.

A Family of Systematic (Code) Reading Techniques

Dieter Rombach
CS Department, University of Kaiserslautern
Kaiserslautern, Germany
&
Fraunhofer Institute for Experimental Software Engineering
Kaiserslautern, Germany

Any engineering discipline depends highly on analysis techniques in order to produce artefacts of high quality. In engineering software, analysis takes place OFFLINE in the form of formal verification and informal inspection activities, and ONLINE in the form of testing. In this presentation, the importance of sound READING TECHNIQUES to support inspections systematically is argued, innovative - socalled perspective-based - reading techniques are presented, their tailoring to company-specific contexts via measurement is explained, and improvement effects wrt. rework redurction and early defect detection are reported. The importance of measurement in the process of trans- ferring human-based techniques - such as reading - into practice is high- lighted. It is claimed that no human-based technology will stick under project pressure if people have not been convinced of its benefits. Using the example of reading, it is demonstrated that this "convincing of the benefits" requires a THREE-STEP EXPERIMENTAL PROCESS: (a) controlled experiments to assess the potential of new reading techniques (to take place in a research laboratory environment), (b) semi-controlled experiments to "sell" the techniques to practitioners (to take place during training), and (c) case studies to optimize reading to project goals and characteristics (to take place in real projects). Results from industry show that systematic reading techniques - such as perspective- based reading - are a powerful tool towards achieving high quality. They should be used in combination with formal verification and testing. No single one is THE silver bullet!

Using PVS in Documentation and Inspections

John Rushby
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
Rushby@csl.sri.com
Phone: +1 (415) 859-5456
Fax: +1 (415) 859-2844

PVS is a verification system (that is, a specification language coupled to a highly automated interactive theorem prover) that is normally used to state and prove conjectures concerning safe or correct behavior of algorithms or system specifications.

However, it is also necessary to inspect PVS specifications to ensure that they capture the author's intent. I will describe joint projects with Collins Commercial Avionics in which PVS was used to verify correctness of some of the microcode and microarchitecture for their avionics processors, and in which the PVS specifications were also used as documentation and inspected by engineers. I will also indicate a methodology developed by Collins in which symbolic simulation performed by PVS was used in inspections of the microcode for their Java machine.

There is little point in inspecting for properties that can easily be checked mechanically, so I will also describe projects in which PVS was used to specify and examine properties of some requirements for proposed changes to the flight software for the US Space Shuttle.

Papers describing these experiements are available via
http://www.csl.sri.com/fm.html

Checking Properties of RSML Specifications Using Formal Verification Tools

Jens Ulrik Skakkebæk
Computer Systems Laboratory
Stanford University, USA

This talk presents recent work to apply formal verification tools to check properties of specifications in the Requirements State Machine Language (RSML) by Leveson et al. RSML specifications consist of a number of subsystem automata, possibly executing in parallel. A condition for each transition specifies whether or not it is *enabled*. Each condition is represented by *a table*, representing the condition in a disjunctive normal form.

Previous analysis by Mats Heimdahl et al. have focussed on checking *consistency* and *completeness* properties of conditions for transitions originating in the same state. The conditions of transitions out of a state are consistent if and only if for all pairs of conditions, the two conditions are not true at the same time. In contrast, the conditions of transitions out of a state are *complete* if and only if at least one condition is true at the same time.

This talk demonstrates the application of the Stanford Validity Checker (SVC) by Dill et al. to check such consistency and completeness properties. SVC decides validity of formulas in quantifier free first-order logic with uninterpreted function symbols and includes interpreted theories for linear arithmetic, inequalities, arrays, records, and bitvectors. It is shown how SVC without modifications can be applied to check properties of a case study which includes linear arithmetic and inequalities. Furthermore, it is demonstrated how the PVS theorem prover of SRI International can be employed in the process to handle non-linear arithmetic and quantified expressions before calling SVC. The work is an illustration that existing general-purpose formal verification tools can be applied to checking properties of table based specifications with modest effort.

Problems with specifying system requirements that are both abstract and practical for validating a software implementation

Lyne Tougas
Safety Evaluation Division "Engineering"
Atomic Energy Control Board (AECB)
Ottawa, Canada

The Atomic Energy Control Board (AECB) developed and distributed for comments a draft Consultative document on "Software in Protection and Control Systems". This document defines what licensees should provide to the AECB as evidence of completeness, correctness and safety of new software - new software referring to software in new systems, software that implements new functional requirements in existing systems, and new software that replaces existing functionality. The presentation focussed on the regulatory requirements concerning the software requirement specifications, the systematic inspection of software design and implementation, the software testing, and evidence of good software process.

The main attributes of the software requirement specifications are unambiguity, correctness, consistency, completeness and reviewability. These attributes should also be met by the system requirement specifications to assure proper verification and validation of the system. It has been found, by the AECB in recent assessments of upgrades to existent systems, that although software requirements specifications included the definition all inputs and outputs and the specification of the relationship between these inputs and outputs, the first versions of system requirement specifications did not always include sufficient information and did not include a complete description of what the system should do. The system requirements documents were then revised to correct these deficiencies.

Algebraic Methods and Mechanical Software Verification

Martin Von Mohrenschildt

As software takes increasingly control of many aspects of your life, e.g. bank, air-plain, car, nuclear power-plant, the correct behavior of this software, the behavior according to its specification, is very crucial if not fundamental. The verification of software, verifying if the code satisfies its specification, is a complex and resource consuming process that has to be better understood and automated as match as possible.

Algebraic software specifications offer a mathematical and precise way to define the requirements and the design of software. But formal methods are hard to work with, often the size of the formulas is quite large. The advantage of formal methods is there processability by computers to automate this verification. We propose to adapt and create computer algebra methods to assist and to automate the verification process. By defining an algebraic structure on tabular expression we are able to compose, simplify and decide equality of certain classes of algebraic specifications.