BCS/FACS CONFERENCE

ALVEY SIG-FM

of the Art Workshop

I M P E R I A L C O L L E G E L O N D O N

16th and 17th December 1985

Location: Lecture Theatre 201, Civil Engineering

TIMETABLE

Day 1 - Monday		Day 2 - Tuesday	
9.30 - 10.15	Registration	9.~~30~~ 15 - 10.15	Cunningham
10.15 - 10.30	C O F F E E		
10.30 - 11.45	Cottam 1	10.30 - 11.45	Cottam 2
11.45 - 1.00	Coleman 1	11.45 - 1.00	Coleman 2
1.00 - 2.30	L U N C H		
2.30 - 3.45	Gaudel 1	2.30 - 3.45	Gaudel 2
3.45 - 4.15	T E A		
4.15 - 5.30	Knowles 1	4.15 - 5.30	Knowles 2

Mme GAUDEL

HOW TO STRUCTURE ALGEBRAIC SPECIFICATIONS: A SPECIFICATION LANGUAGE AND ITS ENVIRONMENT

A specification is supposed to describe a future or existing system in such a way that the properties of the system (what the system does) are expressed, and the implementation details (how it is done) are omitted. Thus a specification language aims at describing classes of possible implementations. In contrast a programming language aims at describing specific implementations.

Among the current formal approaches for specifications, algebraic specifications are especially appropriate for this purpose: the presentation of an algebraic specification defines a class of algebras (also called models). These last words should not frighten the non specialists in algebra since they mean nothing more than some operations on various domains of values. An algebra is just a possible implementation of some operation names and some domain names.

This lecture is an attempt to summarize some basic concepts of algebraic specifications in an hopefully understandable and intuitive way. The support of this presentation is the specification language PLUSS. PLUSS allows to structure large algebraic specifications. Its design is based upon several experimentations on real large examples. It aims at improving the acceptability of formal specifications in industrial context.

The ASSPEGIQUE specification environment supports a subset of the specification language PLUSS This environment is based upon the CIGALE incremental parser constructor. The main capabilities of CIGALE are: simplicity of the operators definition and incremental construction of grammars; efficient handling of modularity (it is possible to add or to remove (sub)languages in the current parsing environment); complete support of specific notions such as coercion and overloading of operators.

The tools integrated in ASSPEGIQUE range from a high-level syntax-directed editor to a symbolic evaluator and theorem proving tools. Therefore ASSPEGIQUE is especially well suited for prototyping purposes and specification debugging. However some other functionnalities such as assisted program construction or test data generation are also available.

Mme GAUDEL

TEST SETS GENERATION FROM ALGEBRAIC SPECIFICATIONS USING LOGIC PROGRAMMING

Functional or "black-box" testing has been recognized for a long time as an important aspect of software validation. It is especially important for large-sized, long-lived systems for which successive versions have to be delivered. In this case, non-regression testing may be long, difficulty and expensive. It should depend only on the functional specifications of the system.

However, most of the studies on test data generatio have focused on program dependent testing, since it was possible to use the properties of a formal object; the program. Of course such an approach is necessary but not sufficient. The emergence of formal specification methods makes it possible to found functional testing on a rigorous basis. In this lecture we present a method and a tool for generating test sets from algebraic data type specifications. We consider hierarchical, positive conditional specifications with preconditions such as the specifications written in PLUSS. We study how to test an implementation of a data type against aproperty (an axiom) which is required by the specification. The formal specification is used as a guideline to produce relevant test data.

The first part of this lecture is devoted to precise and formal definitions of several concepts; first we give the fundamental properties of what we call a collection of test sets; then we state the hypotheses which are assumed during the testing process and ensure the acceptability of the considered collections of test sets. This notion of acceptability is defined and discussed with respect to the classical properties required for test selection criteria: reliability, validity and lack of bias.

In the second part, we show that using algebraic data types allows the introduction of further hypotheses and enables the test sets generation.

The third part describes how to implement and improve this method using PROLOG. It turns out that PROLOG is a well suited tool for generating test sets in this context. In particular it automatically provides partitions of the domains of the variables. Of course, the use of PROLOG somewhat limit the kinds of strategies which can be used to generate test data. Thus two extensions of PROLOG are used: METALOG which allows an explicit control of the traversal of the resolution tree; SLOG which deals with equality and allows the pruning of fruitless computation branches.

DR F KNOWLES

A DISCUSSION OF FORMAL MODELS

This talk is divided into two parts.

The first part is an exposition of the Bell - La Padula (B-L) model. The model referred to is that described in the MITRE report, "Unified Exposition and Multics Interpretation." This model has been translated into a VERSUS specification, and VERSUS versions of some model constructs are presented.

The second part is a sketch of a model (not formal) for a B1 level UNIX.

The B-L model is the most discussed security model for operating systems, so it is appropriate to start with it if one is interested in writing a model for a secure system. In a sentence, the B-L model is a state machine model for an operating system with four access modes (read, write, read-and-write, execute), eight generic state transformations (give-permission, rescind-permission, get-access, release-access, create-object, delete-object, change-current-subject-level, change-object-level) and three state invariants (simple-security, star-security, discretionary-security).

The model is not intended to address covert channels or system integrity, nevertheless it is a reasonable first step in describing those aspects of an operating system that could result in the undesirable transfer of information.

Some aspects of the model are unclear. Giving and taking of permissions is incompletely specified and the rules regarding permissions are stated differently in different parts of the paper.

The rules for accessing objects in a hierarchy are unsatisfactory. They require a user to change security levels after creating an object in order to gain read-and-write access to it.

The proposed model for a B1 level UNIX limits transfer of information between objects of different classification levels in the spirit of the B-L model, but equal emphasis is given to trusted subjects and access restrictions that guarantee system integrity.

DR F KNOWLES

AN EXPERIMENT IN CODE-LEVEL VERIFICATION

This talk describes a research effort to investigate theorem proving requirements for code level verification.

The programming language used is the sequential subset of Toronto Concurrent Euclid, developed at the University of Toronto, Canada. The Verification Condition Generator was written by Phillip Matthews as a Master's Degree project at the University of Toronto. The run-time-specific proof rules are based on those in a Doctor's thesis by W. David Elliott, done also at Toronto. The theorem prover was written by Dam Putnam as part of Compion's (now Gould Computer Systems - Urbana) specification system, VERUS.

The immediate goal is to characterize the verification conditions that can be proven without significant interaction between the user and the theorem prover.

Two hypotheses are being investigated. The first is that verification conditions showing merely the absence of run-time errors fall into this category. The second is that Putnam's prover is especially good at shallow proofs of the type that will be encountered.

New proof rules will be presented that combine a modified form of the "forward" proof rules of the verification condition generator with Elliott's rules for detection of run-time errors.

The verification condition generator has not been modified to work with the new rules, so verification conditions being investigated must be manually generated and given to the theorem prover in the form of a VERUS proof outline.

Examples of programs, verification conditions, proofs, and problems will be presented.

DR I COTTAM

VDM AND THE MULE SYSTEM

LECTURE 1

This lecture will present the dialect of the Vienna development Method (VDM [1]) taught, and in use, at the University of Manchester. An experimental prototype formal methods support system, known as 'Mule' [2], developed at Manchester by Alan Wills, Tobias Nipkow, and others, will also be discussed.

VDM is both a notation and a method. The specification notation is 'model-oriented'; a relational data base (say) might be specified by a model of the system state which uses mappings from relation names to sets of tuples. The operations of the data base system would be specified using pre-and post-conditions of 'before-and-after' state pairs. The method is a collection of proof rules for checking the correctness of program development steps. Development begins with an abstract specification from which more concrete (or design) specifications are produced by means of data type refinement and operation decomposition. The underlying logic of this VDM dialect has been formalised in [3] and is known as LPF - a Logic of Partial Functions. The preferred proof style, used in the tutorial book [1] and in Mule, is a form of Natural Deduction.

The above topics will be presented via suitable examples.

LECTURE 2

In the second lecture the major meta-language of Mule - a logic programming language known as Graphl - will be presented, again, mainly via examples of how such tools as proof checkers or type checkers are written. Further, a method of generating 'Rapid Prototypes' from VDM specification will be discussed. This involves the translation of specifications into Prolog augmented with libraries of Prolog clauses implementing sets, mappings and other common abstractions used in VDM model-based specifications.

References

[1] C B Jones "Systematic Software Development using VDM" to be published by Prentice-Hall International in early 1986.

[2] I D Cottam et al, "Project Support Environments for Formal Methods" in "Integrated Project Support Environments" ed. John McDermid, pub Peter Peregrinus for IEE, 1985.

[3] H Barringer, J Cheng and C B Jones, "A Logic Covering Undefinedness in Program Proofs", Acta Informatica Vol 21 part 3, 251-269.

MACHINE SUPPORT AND FUNCTIONAL SPECIFICATIONS

From a pragmatic point of view there are several reasons for an interest in formal specification. There is the need to be able to *record the design process*. Currently the only outcome of the design stage is a flat program. All the structural information about the way the program has been developed, which is crucial to its comprehension, is lost. There is also a pressing need to be able to *investigate the design ahead of producing the code*.

The complexity of the programmming task, which typically comprises a huge number of small shallow steps, makes machine support for the design process essential. Formality is essential to cracking this problem since in essence there is an equivalence between what is *formal* and what is *machine supportable*.

A formal specification of the functional behaviour of a program must be built on the theories of the data types over which it computes. Capturing the intended functionality is crucial since it is the cornerstone of what is meant by correctness. *Functional specification* languages like OBJ provide a means for abstractly defining data types and operations over them, they can describe the transformations to be effected by a program but can say nothing about the location of results in a program's state.

OBJ can play a number of roles in the software design process.

It serves as a vehicle for understanding the problem to be solved. Its non-executable part can be used to define requirements against which the executable modules may be validated. In this way simple machine support, in the form of type checking and execution, can be used to explore the problem domain quickly and rigorously. The result of this activity is a *validated* and *executable functional model* of parts of the problem which are of interest.

OBJ specifications also contribute to program specification since the values stored and manipulated in the state of a program are concrete representations of the data types introduced in the program's functional specification. Thus OBJ provides a *vocabulary for use in pre-post order specifications* of program components.

Finally the executable modules of an OBJ specification stand as *blueprints for program code*, thus enabling initial versions of programs to be constructed systematically from specifications.

In these lectures we explore these issues and consider the possibility of other kinds of machine support for software engineering based on functional specification languages.

D. Coleman