OOPSLA ’93 Workshop

Behavioral Specification Techniques

Position Paper

Prof. Alfred Strohmeier , Gabriel Eckert
Swiss Federal Institute of Technology in Lausanne

Contact:

Alfred Strohmeier
Software Engineering Laboratory
Swiss Federal Institute of Technology
EPFL-DI-LGL
CH–1015 Lausanne
Switzerland

Phone: +41 21 693 4231
Fax : +41 21 693 5079
E-mail: Alfred.Strohmeier@di.epfl.ch

Abstract:

This position paper provides a quick overview of the various behavioral specification techniques, from decision tables, finite state machines, Petri nets, and program design languages, to formal specification languages. It contrasts property based behavioral models (and languages) with model-based ones.

Introduction

Simply stated a specification is a complete description of the entire external interface of the system to be developed. This includes two types of requirements: behavioral and non-behavioral [Davis 90].

Behavioral requirements define what the system does by describing all the inputs and outputs to and from the system as well as how they interrelate (transform function or operational specifications).

Specifying Behavioral Requirements

As systems become more complex, it is increasingly difficult to explain their behavior in an unambiguous manner, especially when using natural languages. A more specialized language can provide us with a richer, higher level, and more semantically precise set of constructs.

There are a number of specification schools that therefore advocate extreme levels of formality in behavioral specifications. However, we should keep in mind that the first purpose of a specification is to provide a means of communication among all parties. In many case, it is likely that formality acts against readability and therefore does not necessarily bridge the gap between developers and customers.

Behavioral Specification Techniques

The following subsections briefly discuss various behavioral specification techniques. No single approach may be considered sufficient because specific techniques usually only address some aspects of some problems.

Decision Tables

Decision tables are used to relate sets of conditions to prescribed actions. They are easy to understand and may also be graphically represented using a kind of flow chart. Decision tables are only appropriate for those parts of an application which are decision-intensive.

Finite state machines

Finite state machines can only be in one of a given number of states at any specific time. The outputs and the next state are generated by a pure function of the inputs and the current state. Finite state machines can be defined by state transition diagrams or by state transition matrices. Both can be used to describe a Mealy or a Moore machine.

In an object-oriented model, it is straightforward to associate a finite state machine to each object. We believe that this attempt is not very practical: state transition diagrams are quite limited in expressive power and they rapidly suffer from a combinatorial explosion problem.

Statecharts

Statecharts [Harel 87] are extensions to finite state machines which facilitate hierarchical decompositions and allow communication between concurrent finite state machines. It seems that statecharts are a successful attempt to avoid the mentioned problems of finite state machines. Unfortunately, we have no experience of applying them.

Petri Nets

Petri nets are abstract virtual machines that are usually represented as graphs of places and transitions interconnected by arrows. Tokens move from place to place according to well-defined rules.

Petri nets are relatively simple to understand and well-suited to the precise description of process synchrony. However, their use for general requirements specification is less obvious. In particular, Petri nets are probably too low level for most applications.

Program Design Languages

Program Design Languages (PDL or pseudo code) were first introduced for the detailed design of software modules but they can also be applied for behavioral specifications. For instance, decision tables can easily be converted into pseudo code. There is however a high risk of falling into design when using such a formalism. Typically using pseudo code forces us to state a particular order when one is not really required. Algorithms can easily be defined using a PDL, however an algorithm is an implementation rather than a requirement.

Formal Specification Languages

VDM [Jones 90] and Z [Spivey 89] are probably the more popular formal languages used for the specification of sequential systems. Concurrent system specification techniques such as CSP [Hoare 85] and CCS [Milner 80] are not as mature as sequential techniques.

We favor the use of Z because its authors have paid particular attention to its presentational aspects. Z schemas are small chunks, relatively easy to read owing to graphical highlighting. Moreover, a variety of ways of combining schemas are provided (schema calculus). We think that schema calculus is a significant contribution, both notational and conceptual, to our ability to separate concerns within Z specifications.

Formal software specifications are mathematical entities for which it may be possible to prove consistency and completeness. The ability to reason about specifications is one of the advantages of using a formal language. In our opinion, the best reason for developing formal specifications is the deep understanding gained by the developers. This certainly reduces omissions and errors in the specifications.

Unfortunately, clients are unlikely to understand formal specifications. Hall suggests that specifications can be understood by clients by paraphrasing them in natural language and by specification animation [Hall 90].

Another difficult aspect is avoiding the internal design of the software. It is probably natural to define the external product behavior by describing its internal components and this is the fundamental approach used in model-based specification techniques like VDM or Z. However, the definition of an explicit internal structure may easily lead to premature and overconstraining specifications, a very real risk with model-based techniques.

On the other hand, property-based specification techniques (also called algebraic specifications) seem closer to the conventional 'black box' specification method which only describes the required external behavior characteristics while the internal structure is postponed until the design phase.

The algebraic specification approach provides us with a data type theory straight away because the specification **is** the theory (the axioms directly define the properties which show how the operations interact). With a model-based approach, the axioms define behaviors of operations over a model, and the theory has to be constructed from this by appropriate proofs. We consider that property-based specification comes before model-based specification (the former provides us with theories that we try to 'implement' with the latter). Starting with model-based specifications is in a sense working backwards. Unfortunately, algebraic specifications are often less intuitive as it is notoriously difficult to incorporate the natural notion of objects with state.

Z is clearly a good starting point for specifying systems in an object-oriented manner. It is state-based and object operations are easily specified in terms of changes to the state. However there is no simple way of specifying classes and inheritance. Therefore, several object-oriented extensions to Z have been proposed [Stepney 92]. Until now, we have not investigated them.

Operational Specification Languages

Operational Specification Languages allow the construction of executable descriptions. Operational specifications can be executed (or interpreted) and mimic the required behavior of the system to be developed. Executable specifications may also be viewed as an exploratory prototype that is obtained as a by-product of the development process. Prototyping is therefore **inherently** supported.

The structure of an operational specification is problem-oriented, with an explicit internal structure, the external behavior being only implicit in the specification. This is viewed as an expression of the inevitable coupling between **what** is being done and **how** it is being accomplished [Zave 84].

PAISLey [Zave 82] and JSD [Jackson83] are probably the most mature instances of the operational approach. Authors disagree about the 'object-orientedness' of JSD although it is generally recognized that the initial JSD phase (modeling) is close to an object-oriented approach. We argue that the meaning of a JSD entity (object) is primarily an ordered set of actions and not an encapsulation of data and operations.

There are other very high-level languages which allow a concise expression of the problem to be solved. Because of their mathematical basis, fonctional programming languages can also be viewed as a formal system specification. In our opinion, a functional approach to defining the input to output relationship is much less restrictive than a PDL algorithm.

Conclusion

There is little or no formal basis for most work about object-orientation. It is therefore not clear how to incorporate formal specifications of behavior into the object paradigm. The surveyed techniques are all helpful in some way. However no one technique would facilitate the description of all behavioral requirements of a typical application.

References:

[Davis 90] Software Requirements - Analysis & Specification
Alan M.Davis
Prentice-Hall, 1990

[Hall 90] Seven Myths of Formal Methods
A.Hall
IEEE Software
Vol.7 No.5(1990) pp 11-20

[Harel 87] Statecharts: A Visual Formalism for Complex Systems
David Harel
Science of Computer Programming
Vol.8 (1987) pp 231-274

[Hoare 85] Communicating Sequential Processes
C.A.R.Hoare
Prentice-Hall, 1985

[Jackson 83] System Development
Michael A.Jackson
Prentice-Hall 1983

[Jones 90] Systematic Software Development using VDM - Second Edition
Cliff B.Jones
Prentice-Hall, 1990

[Milner 80] A Calculus of Communicating Systems
A.J.Milner
Springer-Verlag, 1980

[Stepney 92] Object Orientation in Z
S.Stepney, R.Barden, D.Cooper (ed)
Workshops in computing
Springer Verlag, 1992

[Spivey 89] The Z Notation: A Reference Manual
J.M.Spivey
Prentice-Hall, 1989

[Zave 82] An Operational Approach to Requirements Specification for Embedded Systems
Pamela Zave
IEEE Transaction on Software Engineering
Vol.8 No.3 (1982) pp 250-269

[Zave 84] The operational versus the conventional approach to software development
Pamela Zave
Communication of the ACM
Vol.27 No.2 (1984) pp 104-118