On The Use of Formal Methods in Software Development

Dines Bjørner
Dept. of Comp.Sci., Techn. Univ. of Denmark
DK-2800 Lyngby, Denmark

Invited Paper*

Abstract

We propose a total framework for the software development stages of specification (definition), design and coding. This framework is based on three cornerstones: (a) the concept of software development graphs which specify all the stages and steps of development; (b) the use of formal methods, in our case VDM, the Vienna Software Development Method, in all stages and steps of development; and (c) the clearly separate rôles of theoretical computer scientists, programmers, software engineers, and development managers in all aspects of software development. Thus not only programming is formalised (ie. programs considered formal objects), but also development, its engineering and management (ie. the entire programming itself is also considered a formal object about which to reason).

Personal Prelude

I have been asked to relate 14 years of experience in "using formalisms in software engineering". I have chosen to tackle this by proposing, as announced in the abstract, a total framework for the development of software. We identify as many of the equally important factors which enter into development and map each of these aspects onto our model for software development. We do not use formalisms for the sake of being just formal. We use formalisms (1) because they appear to help structure more finely the development, (2) because they are the primary means we know of to help guarantee correctness of software, (3) because developments that have used formalisms have been far more productive, by a 3-5 fold order of magnitude, than developments not using formalisms, and (4) because it is fun, including intellectually satisfying, to use formalism. An intrinsic part of the formalism (VDM) we have developed and used is abstraction. Abstraction in defining (specifying) software, but also abstraction in its design. That is: (1) first (specification) we abstract from any implementation and concentrate only on the [properties of the] functions we wish to exhibit to the user; (2) then (abstract design) we abstract from how the software achieves these functions by focusing our attention on what the conceptual software components compute; (3) then (concrete design) we abstractly describe how the software computes, before (4) we finally code the software. Abstraction, and at these various levels, helps (1) "divide and conquer" the [seeming intrinsic] complexity of software, helps constructing (2) conceptually more transparent system implementations. Perhaps a few people can develop as beautiful systems without using formalisms (although I doubt it), but we are here to develop methods for other than the unique geniusses—and, when all is said and done, only formal development can possibly give any trust in correctness of the software system.

The paper is not technical. (1) The short time given (Dec.-Jan. 86) for writing the paper, (2) four weeks of lecturing commitments in this period (India, China, Japan and the USA), (3) the fact that three other papers also had to be written, (4) Christmas and New Year, and (5) spring course lecture preparation—all that may be a bad excuse. But I shall anyway beg the readers understanding. I do, of course, believe that I have touched, in this paper, upon crucial aspects of "the use of formalisms in software engineering". I have chosen to identify the very many facets, aspects and concepts that go into software development and I try to single out those that can already be formalized, presenting the whole in such a way as to invite further for-

*The Use of Formalisms in Software Engineering malization. I am not postulating unproven ideas. All of what is described below has been tried and shown desirable—only we still lack formal tools for their support.

Summary

(Parts i-iv below are found in sections 1-6.)

There are six parts to this paper. The first five (i-v) parts constitute more-or-less independent units which by their being strung together paves the way for the last part: (vi) some reflections on what it takes to get the ideas generally propagated—and what hindrances there might be to this! First (i) we define the concepts of 'theoretical computer science', 'computing science' (or: 'programming methodology'), 'software engineering' and 'software development management' and discuss briefly the rôles of the following categories of software development personel: 'resident scientists', 'programmers', 'software engineers', and 'managers'. Then (ii) we define the concept of 'formal methods' and comment on the state-of-the-art of the use of formal methods in professional programming. We also define the kind of software projects and products to which formal methods are applicable. Now (iii) follows a central part of the paper in which we introduce the concept of software development graphs: these are acyclic, directed graphs whose nodes in general denote specification, design or coding activities leading to documents and whose edges in general denote activities motivating and justifying designs wrt. specifications, code wrt. design, etc. Nodes with no input edges or "early" in the graph denote specification; nodes with no output edges, or "late" in the graph denote coding; interior nodes denote design. This concept of software development graphs is then (iv) discussed from several points of view: first from the quadruple facets of theoretical computer science, programming, software engineering and management; then (v) from the point of view of its implications wrt. a total, not just syntax and pragmatics oriented, but also a semantics oriented software development support environment. The conclusion (vi) reviews the reality, "philosophy" and sociology of getting people to use formal methods.

Introduction

The abstract and summary has outlined our goal. In this introduction let us motivate and justify our approach.

Although the terms 'software crisis' and 'software engineering' seem to have been widely adapted as from 1969 [Randell 69] we see little change, in industry (software houses, computer manufacturers programming centers), in the way of improvement. Most developments of for example Ada[®] compilers went way above initial estimates, did not use any, or only "randomly", some of the very many formal techniques now available, and, as a consequence, it seems, lead to unreliable products which are difficult and costly to maintain.

In the years since 1969 a great number of formal techniques (by some even called methods) have been proposed, almost exclusively in academia—with very few of them actually being used in industry. We are here referring first to such techniques as (1) proving coded programs correct wrt. program annotations in the form of assertions using some form of Hoare Logic (Proof System), (2) formally defining software functions using either algebraic or denotational semantics, and (3) transforming programs from their specifications, via their design, into code. Within each of the above three areas (1-2-3) there is today an abundance of theory and also proposals for its practical exploitation. But, coming to the second part of the first sentence of this paragraph, very little of this is actually being used by industry.

The reasons for this lack of adoption of formal techniques by industry seems not just to be a sociological one, but as well the lack of a firm total method that offers some unifying framework for at least some of the techniques. The sociological reason for lack of acceptance seems rooted in a number of factors: program development managers belong to a generation who were either not taught computer science and programming, or were taught it in ways very different from what these more recent, formal techniques assume—which may again be different from the way the programmers, whom the managers are supposed to lead, were taught. The result is basically that a lowest common denominator concerning software development subconsciously emerges. In this there is no room for formal techniques. The gap between formal techniques offered by research and the "techniques" of industrial software development is widening!

In contrast to this we see the existence of more and larger computer science departments producing not only interesting computer science results, and many candidates formally trained in these, but also, I claim, an undersupply of results in programming methodology, especially as concerns the 'method' aspects of putting 'techniques' together.

It is a telling, at times a disturbing, sign that most of the so-called 'methods' being propagated were basically created in smaller software houses, and by individuals with little or no inclination towards considering a formal basis for their work. Thus most of

[®]Ada is a registered trademark of the US Government (Ada Joint Programme Office) these methods (with the refreshing, notable exception of JSP/JSD [Jackson 75,Jackson 83]) were not arrived at in the open, peer reviews environment offered by academia, but in an industrial one driven by commercial motives. The revolutionary long range, far horizon possibilities of formal, well-founded—openly critisized—university research has been replaced by the evolutionary, short range, here-and-now offerings of ad hoc—at times even zealously religious–proposals.

It is, on the background of the fanfare-like claims of above, therefore maybe a bit presumptious to now state that the goal of this paper is to narrow the gap between the offentimes exotically beautiful semantic results of academia and the always pedantic syntactic and pragmatic concerns of industry: in short to propose a framework, a method for "total" development, where 'total' is taken to mean: the full span from functional specification to coding, the full spectrum from managers, via software engineers and programmers, to project 'resident' scientists, and the full force of all applicable theoretical results applied to the entire span and spectrum.

1 The Computation Sciences and The Rôles of Developers

In this section we take a look at the professions and the professionals of our field (of software development). The aim is to provide a more refined understanding of the internal setting in which software development takes place. [The external setting, having to do with customer relations, will not be dealt with here.] We believe that a more finely grained decomposition of sciences and engineering, and the rôles of people outlined below, is beneficial to development. We cannot, obviously, prove this, but we can refer to developments [Oest 86] which, embodying the spirit of this decomposition, have been not only successful in leading to trustworthy software developed in time and within resources—all at a level far better than compeditive developments, but which were also "fun" projects: intellectually satisfying, and projects in which all the staff had confidence. We attribute part of these successes to an understanding of the aspects treated in this section. We attribute remaining parts to having used a formal 'method' (in this case VDM), and to having formed an odyssée of four years of development around a clearly accepted software development graph. But first things first.

1.1 The Computation Sciences

The computation sciences and engineering consists here of: computer sciences, computing science, software- and hardware [computer] engineering.

[Theoretical] Computer Science

Theoretical computer science is the study of programs:

—of what is computable (meta-mathematics, computability theory, ...), of how complex it is to compute things (algorithm analysis, complexity theory), of the mathematical foundation for various abstractions of computing (automata theory, formal language theory, net theory, fix point domain theory, denotational semantics, algebraic semantics), and of the foundation of the reasoning that goes on in programming (proof systems, Hoare Logics, ...), etc.

Computing Science — or Programming Science and Methodology

Computing science is the study of programming:

—of the methodologies, languages, techniques and tools that go into the development of software: requirements analysis and definition; functional and non-functional specification; program transformation; of the special development techniques required in order to secure reliable, robust, fault-tolerant, and secure software; and coding.

Software Engineering

Software engineering is the practice of programs and programming:

—thus it includes the syntactic and pragmatic human as well as mechanical (tool-building and tool use) concerns of how to produce, validate, control and monitor documents needed in the software development process, not just from the point of view of supporting these processes mechanically, but also of securing interaction (the social processes) between people: customers/users and developers, and between developers.

We thus define the concept of Software Engineering more narrowly than is done usually. The IEEE definition, for example, encompasses our Computing Science.

1.2 Developer Rôles

Programmer

When a person is concerned with the **semantic** aspects of the programming process, for example: which functional properties to capture in abstract specification and which to focus on in concrete design and coding, which formulation to give development documents, what they mean, how to verify what documents describe, and how to transform abstract descriptions into more concrete ones, then that person is a programmer, and is programming. A programmer proposes theories (with the theoretical computer scientist guaranteering these to exist).

Software Engineer

When a person is concerned with the **syntactic** and **pragmatic** aspects of the software development process, for example: with the non-functional, current technology constraints, the tracking of external design [constraint] requirements, the journalling of development documents, the creation and maintenance of versions of specification, design and code documents, their configuration into products, and with the validation of non-functional requirements, then that person is a software engineer. The software engineer is a tool builder (with the tools always reflecting a new instance of a methodology, but bound by current technology constraints and theoretical know-how).

Programmer vs. Software Engineer

The software engineer "harnesses" the laws of nature, the programmer the laws of mathematics, computer and computing science. The software engineer interfaces with the ever current limitations of hardware technology, the programmer with the always fixed laws of computability. The same person is at times a programmer, at times a software engineer, and a good programmer knows exactly when transitions are made between the two activities, and when the assistance of a computer scientist is called for—to help secure the foundations of what is being described.

2 On Software Development and on Software Systems

This section has two parts. In sections (2.1-2-3) we speak of formal methods, and in sections (2.4-5-6) we speak of the properties and qualities of software products and development projects, the object and carrier of the formal methods.

The aim of this section is first to define (sect. 2.1) the concepts of 'method' and of 'formal method' as they relate to software development, to motivate and justify (sect. 2.2) why it is necessary to use formal methods (techniques and tools) whereever possible in all facets of software development, and (sect. 2.3) to briefly nominate one 'formal method' candidate that has stood the test of being applied widely, in Europe, in industrial environments. Maybe not in full formality, but then systematically. The first section (2.1) will therefore define the additional modifiers: 'systematic' and 'rigorous', and section (2.3) will briefly outline what, in the form of the DDC/STC/NBB European Common Market ESPRIT 315 RAISE project [Meiling 86], is being done to offer a method that will satisfy more of the industrial software development needs as they are outlined in the second part of this section.

In section (2.4) we list the properties of software (systems) that the formal methods must cater for, and in section (2.5) the qualities of respectively the project and its resulting software product. Finally, in section 2.6, we itemize some, but not all those informal components that seem necessary, in addition to using formal methods, to help secure the properties and qualities.

2.1 On Systematic, Rigorous and Formal Methods

By a 'method' we shall understand a set of procedures (guidelines) for selecting and sequencing the use of 'techniques' and 'tools' in order to construct an certain artifact. By a 'formal method' we shall understand a method (i) in which all of its constituent techniques and tools are formal, ie. is given a precise mathematical meaning, and (ii) in which the use of the methods and the techniques can be justified formally. We elaborate on pt. (ii). To be formal it must be possible to reason mathematically (logically) about the programs produced (be they abstract, in the form of definitions or specifications, or less abstract, say in the form of designs, or be they executable, in the form of code)—hence the requirement for formal techniques which are thought of as applying to individual, or adjacent pairs of steps of development. But that is not enough. To be formal it must also be possible to reason about the development itself—as we shall see in section 4—hence the requirement for formal methods, where the method part: the orderly selection and sequencing, speaks about programming. Thus we consider not only programs as formal objects, but also programming, the process, is seen as a formal object, likewise subject to reasoning. Tools are such things as the notational systems (specification and design languages) used, and the mechanical aids needed to support clerically (syntactically), semantically and pragmatically the use of the method and its techniques. By techniques we understand amongst other such things as by what principles do we (1) specify abstract definitions, (2) transform these into designs, and designs into code, (3) prove such transformations correct, and (4) discharge other proof obligations in general.

The tools to support the techniques and the method must hence be formally explicable. That is the overall architecture of the full tool support system must transpire from formally expressible properties of the method, and likewise for sub-tools vs. techniques.

We say that a development is formal if all steps of development: definition, design and coding are carried out formally, and if all proof obligations arising from this development are discharged.

We say that a development is rigorous if it lacks being formal by the absence of actual, formal proofs. That is: we are rigorous when we follow the method steps: specify, design and code, and when we formally relate development stages in the form, for example of injection relations, abstraction functions, adequacy or implementability predicates, and theorems of correctness, but omit the detailed proofs.

We say that a development is systematic if it lacks being rigorous by the omission of formal relations between stages of development.

(Our distinction between 'rigorous' and 'systematic' is not consistent with the usage of the same words in [Jones 80,Jones 86].)

Thus, to us, the spectrum formal- rigorous-systematic, is one within which we wish our development to take place—supported, whenever possible by the availability of formal tools.

2.2 Justifying Formal Methods

Before costly software development, like for example that of compilers, operating systems, database management systems, local and wide area nets and distributed systems, incl.. components of ISO/OSI, before development of such software takes place we assume (as a dogma) the creation of a specification. Just like the architects conceptions and drawings of a building, a specification serves (1) as a "legal" contract between customer and developer, (2) as a basis for development, and (3) as a basis for the writing of customer/user manuals. Of a specification we expect that it be (4) consistent and complete, (5) short and concise, (6) accessible and referenceable, and (7) so expressed that formal properties of resulting software can be proved wrt. the specification—an example is: correctness. As a consequence of especially requirements (1,2,4,5,7) we conclude that the specification must be formalized—albeit in such a way that requirements (3) and (6) will be met.

[Bjørner 85a] elaborates on the points made above in the light of the DDC/CRAI/CNR and Genoa University development of a full, formal mathematical definition of Ada.

2.3 One "Formal" Method

The VDM (Vienna Development Method) first emerged in the IBM Vienna Laboratory around 1973-74 (through the work on constructing a compiler for PL/I [Bekić 74]). VDM was subsequently further developed as witnessed, for example by the text books and monographs: [Bjørner & Jones 78,Jones 80,Bjørner & Jones 82,Jone A set of 5-6 volumes on all aspects of VDM is presently being readied for publication by 1988 [Bjørner 88].

VDM has been used in many development projects, mostly only systematically. The most notable such development must undoubtedly be the Dansk Datamatik Center development of the DDC Ada Compiler System [Bjørner & Oest 80,Oest 86]. A growing number of european companies are using one or another aspects of VDM. As a consequence of this (and, it may be said, the success of the DDC Ada Compiler), the EEC (European Economic Community) has established a VDM-Europe Forum whose industrial and academic members meets 3-4 times annually to discuss issues of (1) experience in the use of VDM and its applications, (2) training and education requirements, (3) tools, (4) formal foundations and (5) facets of VDM potentially subject to industry standardisation. The VDM-Europe Forum held its first open symposium March 23-26, 1987 [Bjørner et al. 87].

VDM is basically a Denotational Semantics (ie. model-theoretic, constructive/based method. Facets of VDM is (1) its wide-spectrum specification and design language: Meta-IV [Bjørner & Jones 78], (2) a great number of operation decomposition and data reification techniques [Jones 80,Jones 86]. Meta-IV, in its full extent, is not a formal language, but large subsets of Meta-IV can, or have, been given a formal semantics, and parts have been given a formal proof system (see papers in [Bjørner et al. 87]). VDM is applicable to the specification, design and coding of deterministic systems. Some applications have been made to non-deterministic systems, and operational extensions of Meta-IV, to include CSP features [Folkjær 80] have been used to develop concurrent and distributed systems.

But VDM, in its present stage, is not good enough! One wants full formality, or at least to strive for far more formality in the use of VDM. The DDC/STC/NBB RAISE project sponsored in part by the EEC ESPRIT programme attempts to remedy the wider range shortcomings of VDM. The RAISE Specification and Design Language features (1) algebraic specification and design structuring facilities (abstract data types and data type operations: enrich, derive, combine, abstract and apply), (2) concurrency in the form of trace assertions and CSP, (3) a full, formal semantics, and (4) a proof system. The RAISE Method offers a comprehensive set of development refinement principles—centered around a fully developed data model for all stages and facets of development be they of concerns to programmers, software engineers or project management. Reference [Meiling 86] offers a comprehensive introduction to RAISE while [Bjørner 85b] offers a rôle and scope (requirements oriented) view of RAISE.

2.4 Software Properties

We wish to develop software for (a) deterministic as well as (b) non-deterministic, (c) sequential as well as (d) concurrent systems, and for systems that function in (e) real-time and are (f) distributed. We wish the software for these systems to be (1) fit for their purpose, (2) correct wrt. specifications, (3) reliable, (4) fault-tolerant, (5) secure, (6) maintainable, and (7) robust. Reliable software clearly rejects in-data not specified as input. Fault Tolerant software repairs or "corrects" erroneous in-data (to "nearest" specifiable input) and also detects and corrects or by-passes spurious, intermittent changes in stored data (or program). Secure software systems prevent un-authorized users from finding out (i) what the systems are doing, and (ii) how they are doing it. In addition a secure system prevents un-authorized users from (iii) preventing the system to do what it is doing, and (iv) leaves them not knowing whether they know (i-ii-iii-iv)! Assuming that some measure of a reasonable change in functionality, or other, of a system, its software is said to be maintable if resources required for insertion of these changes can be precisely estimated and in some (here unspecified) way are commensurate with the changes. Software is robust if any changes to it does not alter any of its properties (1-7)!

The usefulness of formalisms in tackling all aspects (1-7) remains to be proven. Certainly aspects (2-3-4) seems separately obtainable [Ravn 86], and aspects (6-7) seems to result from using conceptually clean definitions and homomorphic transformations of these. But there is still a long way to go before we can fully guarantee all aspects fully satisfactorily!

2.5 Quality Assurance

We distinguish between qualities of a project and of its resulting product.

For a project to be a quality we require that it be (1) resourceable and predictable, ie. that, based for example on a complete and consistant software development graph, one can correctly estimate all required resources. people, machines, and finances—month-by-month; (2) controllable: there must be an objective principle by which one can monitor, and if need be, alter consumption of resources; (3) economical: there must be some measure of affinity between budgeting, financing, accounting and (expected) results; (4) secure: the project must lead to expected results; and (5) fun: the project staff must trust that management has full authority and that they are intellectually, educationally and traningwise enriched by the project.

For a product to have quality it must satisfy points (1-7) of section 2.4 above.

Quality assurance is the the twofold act of securing both full project and product quality.

We may be able to formally support achieving product quality. The informal points raised in section 2.5 and the formal one of sections 3-4 are believed to indirectly help achieve project quality.

2.6 Project Components

A number of project components, in addition to the proper, actual specification, design and coding development steps, have been found necessary to help guarantee a high project and product quality. Since they are not normally thought of in the context of software development we list and briefly explain them here.

The first activity is that of defining the project itself. This activity is, in our case, centered around (1) the stepwise development of the software development graph of the project—see sections 3-4. Two activities are then started, activities which run the entire duration of the project: establishing and maintaining, (2) a Terminology, and (3) a library.

2.6.1 Terminology

Today most software [development projects] introduce a number of new concepts, and build as well on previously established concepts not quite fully established. It therefore seems important to create (electronically), maintain and throughout the project to adhere to a Terminology: a set of precise definitions of concepts used and introduced. The importance of establisehing, maintaining (incl. adjusting) and adhering to a Terminology was emphasized, to me, by Peter Naur—whom I hereby gratefully acknowledges. Misunderstandings concerning meanings of names of concepts, vagueness as to their meaning, etc. is most damaging to the pursuit of a project. The discipline of establishing, maintaining and adherence to a Terminology is an intellectually most rewarding (educating and liberating) activity.

In a sense a Terminology is a formal object in that it focuses on precision and economy, ie. conciseness.

2.6.2 Library

In pursuing each step of a project scientific, technical and other papers (reports and publications) are studied: discarded or used. A Library of all such is built, for subsequent reference and for full project documentation. By a Library we understand a three part thing:

(1) a schema which taxonomically structures the bibliography; (2) an annotated bibliography, ie. a list of entries, each not only containing a proper literature reference, but also, as the project goes on, an accumulative set of annotations concerning the disposition of the reference (its value to the project, etc.); and (3) the paper collection itself.

Similarly (to the Terminology) a Library becomes a formal object.

2.6.3 Study-Experiment-Action

Each component of the development proper, to us consists of three phased activities: study-experiment-action. Their resource consumption and time duration typically are 1-2-8. In the first, the study phase, all project component members study what they believe is the literature relevant to the project component at hand. Colloquiae presentations, by all members, identify those ideas and techniques that might be of interest to the project. These are further investigated in the next phase where experiments (in either specification, design, or coding) involving these ideas and techniques are pursued. That is: applied to small, but difficult aspects of the problem. Finally, enough confidence as to the technical choices to take should transpire from the study-experiment phases, and a full scale application to the "real" problem takes place—in this, the action phase.

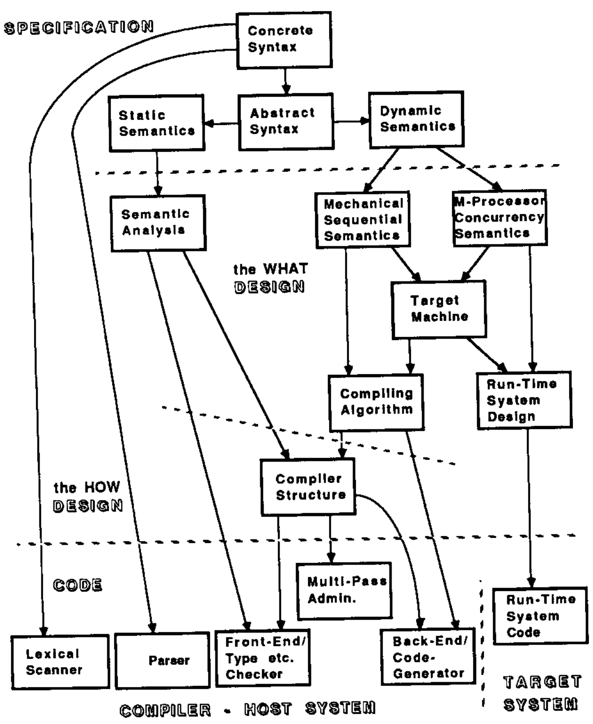
The study-experiment-action scheme is explicitly aimed at furthering the use of formal techniques.

3 A Concrete Software Development Graph

A Software Development Graph [Bjørner 86a], [Bjørner 86b] is, syntactically speaking, a finite, cycle-free directed graph. Nodes denote software development activities which are parts of specification, design and coding. These activities lead to specification, design or coding documents. Edges likewise denote activities which (first) motivates and (subsequently) justifies the step of development designated by the node-edge-node triple identified by the edge. The activities are performed by a combination of programmers, scientists, software engineers and managers. The documents are usually formal.

Software development graphs can be abstract, but meaningless, as in Fig. 3.1, or software subject related, as in Fig. 3.2.

We briefly, for the sake of familiarizing you with concepts of software development graphs, review the latter graph based on a VDM usage.



A Software Development Graph
for Compiler + Runtime Systems
Fig.3.1

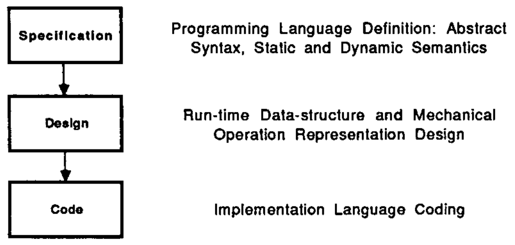

Specification
Design
Code
Programming Language Definition: Abstract Syntax, Static and Dynamic Semantics
Run-time Data-structure and Mechanical Operation Representation Design
Implementation Language Coding

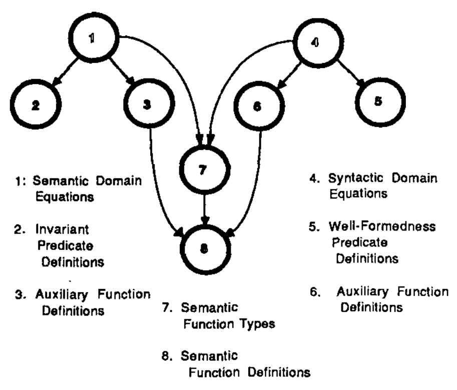
Figure 3.2 : Universal Software Development Graph

We assume a BNF grammar for syntactic Ada given. From it (etc.) we develop an Abstract Syntax, a set of domain equations, in Meta-IV, which abstracts away from such considerations as keywords, linear sequence of objects (like declarations whose linear ordering is accidental and which have no meaning). From the Abstract Syntax we develop, as abstractly as possible, both a Static Semantics, and a Dynamic Semantics, both including treatment of concurrency, etc. Together these four nodes (and their intervening edges) represent the specification part of development. Now follows, in general, a number of design stages. First we focus our development attention on <u>what</u> to do in the (increasingly) more concrete, then on <u>how</u> to perform the so-designed functions. That is: we first develop more concrete, sequentialized version of the Static Semantics (called Semantic Analysis), and more operational, or mechanical, versions of sequential aspects of the Dynamic Semantics (called Mechanical or Macro-Substitution Semantics). The latter exhibits fundamentals of the run-time structure of [Ada] programs. From the Mechanical Dynamic Semantics we first may derive the definition of a Virtual Target Machine, a machine "optimally" suited for executing programs in the source language (viz.: Ada); or such a Target Machine (viz.: DEC VAX) is already given. From the Target Machine definition and the Mechanical or Operational Dynamic semantic Semantics is derived a so-called Compiling Algorithm. This latter defines the exact sequence of target machine instruction to be generated for each (schematic) source language construct. From the concrete what of the front-end (Semantic Analysis) and the back-end (Compiling Algorithm) we develop the, possibly Multi-Pass Structure of the compiler, all the intermediate languages, and the Multi-Pass Administrator. From the Dynamic Semantics is also derived a concrete description of what to handle in tasking; and from this and the Target Machine definition is developed a design for the Run-Time (Target) System. This concludes the design. From respective design nodes is finally developed the actual coding of the compiler and run-time system.

Each of the nodes can, using VDM in general, be refined into a subgraph of eight nodes, see Fig. 3.3.

To each of the nodes and edges correspond a well-defined set of formal techniques.

We explain these, in general, for each of the four areas: programming, engineering, management and theoretical computer science; but first we take a general look at graphs.


1
2
3
4
5
6
7
8
1: Semantic Domain Equations
2. Invariant Predicate Definitions
3. Auxiliary Function Definitions
4. Syntactic Domain Equations
5. Well-Formedness Predicate Definitions
6. Auxiliary Function Definitions
7. Semantic Function Types
8. Semantic Function Definitions

The Sub-Nodes of a single Node Specification or Design

The Denotational Semantics Case

Fig.3.3

4 On Software Development Graphs in General

4.1 Graph Development

Meta-Graphs

To each ("equivalence") class of software there corresponds a meta software development graph. A meta software development graph thus describes how to develop a whole class of "similar" software. The graph above is a meta-graph for the development of compilers of the ALGOL/Pascal class but with tasking (processes, concurrency). Thus it is valid for the development of compilers for such languages as Concurrent Pascal, Pascal Plus, CHILL, occam®, Modual-2, and Ada. Other classes exists: one graph, basically is needed for the development of, say, UNIX operating systems, or LANs (local area nets), or Relational Database Management Systems, etc. Meta-graphs are thus generic and their structure and meaning is the result of research and should be arrived at by university or industry scientists.

Project Graphs

The software industry when faced with the start of development of software selects an appropriate meta-graph from which it develops a project graph. The project graph is a possibly enriched or derived version of the meta-graph together with its four classes of annotations. These four classes are those giving attributes of nodes and edges a-priori defining the tasks ahead for programmers, software engineers, managers and "resident" scientists, as well as a-posteriori remarks describing major properties of the software development, also

®occam is a registered trademark of inmos Ltd. (UK) for each of the four classes, as they transpired from carrying out the project itself.

We thus emphasize the importance of carrying out thoroughly the careful planning manifested by the act of assigning attributes.

Product Graphs

Software engineering spans from requirements tracking, testing to product engineering. For each configuration of versions (see section 4.2) there corresponds a product—and to it a graph, a "version" of the project graph. Instead of certain, and in addition to other project (ie. process) annotations it additionally contains product (ie. result) attributes.

4.2 Programming Attributes

The primary attribute here is that of the method used in development (examples: VDM, HDM, HOS, JSD, RAISE, ...). This attribute pertains to an entire graph. On a node and edge basis we have, referring here only to VDM related) secondary attributes. Nodes: specification techniques such as configuration, or resumption, or exit, or direct semantics—for modelling GOTOs and exceptions; flat location/value, flat location/structured value, or structured location/value storage model; imperative model; applicative model; primarily putative function definitions; primarily implicit (pre-/post-condition) function definitions; etc., etc. Edges: design tranformation techniques such as fold/unfold/abstract/instantiate rule gruded transformations; fully automatic transformation (ie. compiling); or manual transformations—the latter further annotated with summary of theorem proving strategies and tactics; etc., etc. In case of non-VDM based development other attributes may be relevant.

4.3 Software Engineering Attributes

There are here four primary attributes: (1) name and characteristics of the mechanized support system (tools); (2) the requirements tracking principles; (3) the test generation strategy; and (4) the journaling, version and configuration principle. Secondary attributes for each of the respective areas follows trivially.

4.4 Management Attributes

There are two a-priori, primary management attribute inputs: per node and per edge there is an estimate, in the form of a histogram, of per month (virtual time) manpower (machine, etc.) requirements (to do the node, respectively edge activity), and for the entire project (ie. its graph) there is a total available manpower (machine, etc.) histogram. A primary "output" is a mapping of required resources onto real time according to one of a number of allocation principles and as constrained by the availability histogram [Lynenskjold 87].

There are other primary management attributes: principle for project monitoring and control including priority and reallocation schemes, principle for rolling plan (report) generation; etc. Monitoring includes frequency of sampling actually used resources; control stipulates principles for corrective or adjustment actions (for over- or underuse of resources); and the rolling plan is a tabular presentation of these matters intended for use in (1) project management meetings, and (2) accounting.

4.5 Theoretical Computer Science Attributes

Nodes denote documents, and (specification, design, or code) documents denote theories. Edges denote mappings, sometimes morphisms, between theories. The exact nature of the theories and mappings can be planned and the resulting documents checked wrt. these attributes.

Usually specifications define total functions and hence usually requires reasoning in a 2-valued logic. Usually design documents define partial functions and hence requires reasoning in a 3-valued logic. Edges between for example 2- and 3-valued logic "nodes" also denote institution morphisms. Some, usually specification nodes denote documents which define domains in a Scott theory of retracts, whereas other, usually design nodes, denote documents for which a simple (set-theoretic) domain theory á lá Blikle [Blikle 83] suffies. Many other computer science attributes could be listed.

4.6 Remarks

We have neither elaborated on project versus product attributes, nor on those attributes (of nodes and edges) which are assigned during the project proper. Nor have we systematically distinguished between ab initio developer input attributes and partially or fully derived (computed) attributes. There are many other attributes, attribute schemes and properties which we have not dealt with either!

The whole point of assigning attributes and later, in development proper, adhering to their prescriptions, is to formalize as many aspects of development as is reasonable. This "disciplining" gives freedom during actual development.

5 Architecture of a Total Software Development Support System

We have outlined, in section 4, a software development graph based method. We have explained, cursorily, some of its detailed points wrt. the embedded use of VDM. Other formal techniques, different from those of VDM, could instantiate the software development graph based method. Examples of the latter are: JSD ("Jackson System Design"), HDM ("Hierarchical Development Method"), HOS ("Higher Order Software"), RAISE ("Rigorous Approach to Industrial Software Engineering"), etc.

In this section we shall outline the architecture for a total software development support system. By such a system we mean a computerized (hardware/software) system which provides mechanized tools for all aspects of development (specification, design and coding)—both of software development (meta, project, and product) graphs and of the software whose development they prescribe; for all groups of developers (programmers, software engineers, managers and scientists); and which transparently reflects the entire software development graph based method, as well as the particular set of specification, design and coding techniques (like for example VDM) to which it is instantiated.

Thus the architecture should directly, at the outermost, software development graph level, embody the notions of (1) software development graphs—including meta-, project-, and product graphs; (2) their development and maintenance—including attribute assignment and computation; (3) nodes and edges—including their syntactic-, semantic- and pragmatic meanings, for both programmers, software engineers, managers, and resident scientists; (4) the execution of software development graphs—allowing for the concurrent execution of independent nodes and edges, and within each of these the concurrent execution (of a node or edge) by several developers from all of the four developer categories. Points (1-2-3) relate, in a sense, to the development of software development graphs. Point (4) to their execution. By an execution of a software development graph we understand an interaction between developers and the system according to the meaning of graphs, nodes and edges, and as parameterized by the embedded (ie. instantiated) set of techniques (for example VDM).

The below "tree" summarizes the basic taxonomical structure and salient features of the proposed system

Graph Development Sub-System Support

- Graph Repository Support
  (Meta Graph Library)
- Graph Refinement Support
  (Specification and Design)
- Graph Instantiation Support
  (From Meta-, via Project- to Product Graph)
- Attribute Assignment Support
- Attribute Computation Support

Graph Execution Sub-System Support

- Node Executor
  - Syntactic Tools
    * Structure and Syntax Directed Editors
    * Pretty Printers
    * Journaling Support
    * Cross-Reference etc. Support
  - Semantic Tools
    * Data-, Logic- and Control Flow Analysis Tools
    * Static (Type) Checking Tools
    * Dynamic Interpretation Tools
    * Theorem Provers and Verifiers
  - Pragmatic Tools
    * Requirement and Design Decision Tracking Tools
    * Version and Configuration Control Tools
    * Test Case Generators, Testers and Validators
- Edge Executor
  - Automatic Node Derivation Tools
    * Transformation, Unifier and Rewrite Rule Tools
  - "Manual" Node Derivation Tools
    * Structure Editor, Substitutor, Rewrite Rule Tools
    * Proof Obligation and Discharge (Proving and Verifications) Tools

Management Sub-System

- Resource Allocation, Monitoring and Control Support
- Graph Evaluation, Monitoring and Control Support
- Node Evaluation, Monitoring and Control Support
- Edge Evaluation, Monitoring and Control Support

'Resident' Scientist Sub-System

- ...

Some remarks may be in order.

(i) The above "tree" reflects the functional offerings of the systems, not its implemented structure. The fact that there may be a "window manager", a "command shell", and "object oriented data base"—possibly distributed, a "transaction processor"—permitting backouts and recovery, etc., really is of very minor concern to the user. They do not reflect neither the method nor its techniques, rather they reflect implementation (technology) constraints.

(ii) The function "tree" thus reflects that the whole of the system set of tools are deeply rooted in the semantics of the method—rather than merely representing the usual ad hoc assemblage of mostly syntactic and pragmatic tools not bound together by any method.

(iii) We wish the user of the system (the developer) to have, at all times, an exact knowledge and "feel" for where in the development process (ie. where in the graph) he is, and to have access to tools for all semantic, syntactic and pragmatic facets of the method and its techniques. In summary we wish the user to feel like in a driver's seat of a comfortable advanced driving car: wheel, gas pedals, brakes, gear (stick or automatic) shift, and a variety of switches, one each for a great variety of functions. Driving can be a pleasant interaction between car and driver; so should the use of the development support system.

(iv) In toto the conceptual rather than implementational structure of the support system aims at inducing the use of a formal method with all its constituent formal techniques and formal tools.

6 The Reality, Philosophy and Sociology of Using Formalisms in Software Development

We have proposed a total framework for the development of software. We have built this framework around many years of using VDM; and the most basic aspects of the framework has been used for at least 6 years. So we are not postulating new untested, unproven ideas. On the contrary. But we have basically used these ideas systematically, rather than formally, basically because we have not had the necessary computerized support to assist us. We have repeatedly recognized, throughout the years the need for computerized support but have (wisely, we now believe) not rushed into instrumenting ideas which then were not conceptually fully developed. Where before, using no tools, of any of the kinds outlined above (neither syntactic, semantic, nor pragmatic)—where before—we could record development costs only 25% of those of other developments, we expect to do much better with formalized tools! But we do not expect these to result in savings in actual development costs, nor significantly shorter development times. Instead we expect to eventually be able to deliver trustworthy software, software sastisfying all of the product qualities outlined in sections 2.4-2.5. We have become too ingrained with the idea that software development is too costly. I believe it is too cheaply done!

Customers will not rise to demand all of these qualities—although a few (like the US DoD) will ask for some. Instead new software houses will be able to guarantee increasingly more of these qualities—and it is this way around that we shall see the coming of the use of formalisms: through more appropriately fitted products, through more trustworthy software, through higher (relative) productivity, but also because the younger generation of software developers will not want to do it any other way.

There is little hope that existing ("old") software houses and computer manufactures will catch on to the formal methods of developing software. The old ways of doing software are just too ingrained in management and staff. I have experienced in established industry huge gaps in basic knowledge about even the most basic concepts—wrt. what the current graduates are learning. Most existing management is not even themselves able to develop software and thus have no authority to do management and are completely at the mercy of the jargon and excuses of their "programmer-software engineers".

I believe that progress in use of formal methods will only come provided three conditions are met: (1) production of enough candidates from computer science departments oriented towards programming methodology—too many have learned about theories about programs of no use in programming; (2) employment of such people in critical mass groups; and (3) appropriate tools. Production of appropriately educated and trained candidates is the responsibility of universities, and most still fail to understand and distinguish computing science from computer science, and to emphasize computing science! Employment is the realm of software houses, and very, very few of their managers understand to hire appropriately, and formally trained people, let alone assign, the few they get, to form sufficiently staffed groups. Production of tools are, again, in the domain of industry. University produced tools are usually semantically advanced, but prototypes in the senses (1) of being applicable only to very small data (ie. not realistically applicable to the kind of large examples usually arising in industry), and (2) of being poorly documented! Industry produced tools are invariably always semantically void, featuring only syntactic and pragmatic tools, and then in ad hoc combinations.

The old ways in which software is being designed today by IBM and all of its competitors will never be proven wrong. They will eventually just die from lack of use, by new generation having grown up with the new methods:

> "Old theories are never proven wrong. They just die from lack of use by new generations

having grown up with new theories" — *Erwin Schrödinger*, Autobiography.

What we have proposed in this paper is a "theory". There are many "theories". This one, like most other software development "methods", is not a theory in the sense of being provable, ie. provably "the theory". It is, as most computer and computing science "theories", more a conjecture. No proof can be provided for its superiority—nor for others' "theories". Only practice will tell, and control such "change of command":

"There are no theories, and there are no proofs. There may be conjectures, and sometimes there are sad refutations" — essence of one aspect of *Karl Poppers* Theory of Science.

In closing summary: we have tried for each facet and aspect of software and its development to identify ways and means of formalizing these—cf. remarks on formalisms in most sections and subsections!

References

[Bekić 74] H. Bekić et al.: A Formal Definition of PL/I, TR25.139, IBM Laboratory, Vienna, 1974.

[Bjørner & Jones 78] D. Bjørner & C.B. Jones: *The Vienna Development Method: the Meta-Language*, Springer Verlag, Lecture Notes in Computer Science, vol. 61, 1978.

[Bjørner & Oest 80] *Towards a Formal Description of Ada*, Springer Verlag, Lecture Notes in Computer Science, (eds. D. Bjørner & O.N. Oest), vol. 98, 1980.

[Bjørner & Jones 82] D. Bjørner & C.B. Jones: *Formal Specification and Software Development*, Prentice-Hall Intl., 1982.

[Bjørner 85a] D.Bjørner: *The Rôle and Scope of the Formal Definition of Ada*, Dansk Datamatik Center, Techn.Rept. AdaFD/ DDC/ DB/ 6 1985-12-09, 1985.

[Bjørner 85b] D.Bjørner et al.: *The RAISE Project: Fundamental Issues and Requirements*, Dansk Datamatik Center, Techn. Rept., RAISE/DDC/EM/1 v6, 10. Dec. 85, 1985.

[Bjørner 86a] D.Bjørner. *Project Graphs and Meta-Programs: Towards a Theory of Software Development*, Capri Conference on *Innovative Software Factories and Ada*, May 1986, to appear in Springer Lecture Notes in Computer Science, eds. N. Habermann & U. Montanari, 1987.

[Bjørner 86b] D.Bjørner: *Software Development Graphs: A Unifying Theory for Software Development?*, New Delh, 6th Conf. on *Foundations of Software Technology and Theoretical Computer Science*, Springer Lecture Notes in Computer Science, vol. 241, ed. K.Nori, pp. 1-9, 1986.

[Bjørner et al. 87] *VDM Symposium '87 — a Formal Method at Work*, VDM Europe Conference, Brussels, March 87, Springer Verlag, Lecture Notes in Computer Science, (eds. D. Bjørner, C.B. Jones, M. Mac an Airchinnig, E. Neuhold), 1987.

[Bjørner 88] D. Bjørner: *Software Architectures and Programming Systems Design*, 6 volumes, 1988.

[Blikle 83] A.J.Blikle & A.Tarlecki: *Naive Denotational Semantics*, in: *Information Processing 83*, ed. R.E.A. Mason, IFIP Congress Proc., North-Holland Publ., 1983.

[Folkjær 80] P.Folkjær & D.Bjørner. *A Formal Model of a Generalised CSP-like Language*, Information Processing 80 (ed. S.H.Lavington), North-Holland Publ., pp. 95-99, 1980.

[Jackson 75] M.A. Jackson: *Principles of Program Design*, Academic Press, 1975.

[Jackson 83] M.A. Jackson: *System Development*, Prentice-Hall Intl., 1983.

[Jones 80] C.B. Jones: *Software Development, a Rigorous Approach*, Prentice-Hall Intl., 1980.

[Jones 86]	C.B. Jones: *Systematic Development Using the VDM Approach*, Prentice-Hall Intl., 1986.
[Lynenskjold 87]	S. Lynenskjold: *Project Graph based Resource Allocation Scheme*, Techn. Rept., Comp. Sci. Dept., Techn. Univ. of Denmark, Jan.87, 31 pages.
[Meiling 86]	E. Meiling & C.W. George: *The RAISE Language and Method*, Proc. ESPRIT Techn. Week, Sept. 1986, North-Holland Publ., also: Dansk Datamatik Center Techn. Rept. RAISE/ DDC/ EM/ 21 v1, 11. Sept. 1986.
[Oest 86]	O.N. Oest: VDM from Research to Practice, IFIP Congress '86, North-Holland, *Proc. 'Information Processing 86'*, pp. 527-533, 1986.
[Randell 69]	Software Engineering, Rept. NATO Sci. Comm., (NATO Sci. Affairs Div., Brussels 1039, Belgium), Jan. 1969 (eds. P. Naur & B. Randell).
[Ravn 86]	A.P.Ravn: *Fault Classification in Distributed Systems*, in: *Information Processing 86*, ed. H.J.Kugler, IFIP Congress Proceedings, North-Holland Publ., pp. 735-738, 1986.