How Engineering Mathematics can Improve Software

David Lorge Parnas[1],

[1]Middle Road Software, Ottawa, Ontario, Canada

Abstract - *For many decades computer science researchers have promised that the "Formal Methods" developed by computer scientists would bring about a drastic improvement in the quality and cost of software. That improvement has not materialized. We review the reasons for this failure. We then explain the difference between the notations that are used in formal methods and the mathematics that is essential in other areas of Engineering. Finally, we illustrate the ways that Engineering Mathematics can be useful in software projects*

Keywords: Software design, mathematics, formal methods

1 Introduction

There are two basic differences between the field known (euphemistically) as Software Engineering and the traditional Engineering fields such as Electrical, Mechanical and Civil Engineering.

- In Software Engineering, physical science is not as important as it is in other Engineering areas. Knowledge of physical science may be important for specific applications but it is not in the core body of knowledge for software development.

- Mathematics is not used by software developers in the way that it is used in the traditional Engineering disciplines.

The first of these differences is not surprising. Unlike radios, cars, and bridges, software is not a physical product. The second difference, is very surprising. In software, the laws of physics must be replaced by mathematical laws that determine the behaviour of programs [15]. When an Electrical Engineer constructs a device from resistors, inductors, and capacitors, it is routine to calculate the behaviour of the assembled circuit mathematically. Laws describing program composition could be used by software developers in the same way that Kirchoff's laws are used by Electrical Engineers but, they are not; most software developers rely on intuition and trials to determine the behaviour of the constructed program.

2 Roles of mathematics in Engineering

In Engineering, documentation is the design medium [9] and mathematics is used in that documentation. There are two distinct roles for documents, description and specification.

- Documents can be used to describe properties of a product that exists (or previously existed).

- Documents can be used as specifications to state properties that are required of a product. The product specified might not exist.

The difference between a specification and other descriptions is one of intent, not form. Every specification that a product satisfies is a description of that product. The only way to tell if a description is intended to be interpreted as a specification is what is said about the document, not its contents or notation. I consider the phrase "specification language" to be nonsense. Any notation that can be used to produce specifications can also be used to produce descriptions.

Because documents contain mathematical expressions, they can be used to verify the correctness of designs or compute properties of a product.

3 Early approaches to using mathematics in software development

Many computer scientists have proposed ways to use mathematics or mathematical notation to help in program development. It is more than 40 years since the late Robert Floyd [6] showed us how to "assign meaning to programs" and how we could verify that programs will do what they are intended to do. It is at least 35 years since I first heard Jean-Raymond Abrial present the ideas that were the basis of Z and its many dialects. The VDM community began its work about the same time. Dijkstra [4] showed how to use predicate transformation rules shortly after that. Mills (and others) showed us how the classical mathematical concepts of a relation could be used to do the same things [15].

None of these approaches has had the effect on software development practice that was sought. Since 1967, there have been numerous "revolutions" on the hardware side and amazing improvements in man-machine interfaces. The computer systems on my desk today were unimaginable when Bob Floyd wrote his 1967 article. Unfortunately, there hasn't been comparable progress in formal methods. There have been new languages and new logics, but the program design errors we saw in 1967 can still be found in today's software. Paradoxically, successful applications of formal methods to industrial practice remain such exceptions that people write papers about them, thereby confirming that the use of the method is not common.

4 Claims of progress and adoption

The CS research literature reveals that 'formal methods for software development' are a very popular

research area. Variants of the most popular approaches are frequently discussed at conferences and in journals.

Research funding agencies often require larger projects to involve cooperation with industrial organizations and to demonstrate the practicality of an approach on "real" examples. When such efforts are reported in papers, they are almost invariably presented as successful. Paradoxically, these success stories reveal the failure of industry to adopt formal methods as standard procedures; if use of these methods were routine, papers describing successful use could not be published. Industry is so plagued by errors and high maintenance costs that they would be eager to use any method they thought would help; it says something that the "successes" have not been followed up by requiring that the methods be routinely used.

Often, reports of successful industrial adoption do not stand up to scrutiny. Sometimes, the authors are just playing with words. For example, the technique of placing debugging statements in code, which was taught to me in 1959, has recently been trumpeted as "industrial use of assertions".

Close scrutiny of an effort to demonstrate the utility of formal methods often reveals truly heroic efforts to develop and verify complex formal models but little evidence that the actual code is correct. These efforts rarely lead to repeat use or broader adoption of the method.

Some of the reported successful trials may be attributed to the simple method of having two people solving a problem and critiquing each other's code. Thirty years ago, in a paper that is still worth reading today, Elovitz [5] described an experiment in which a program in a programming language was used in the way that formal method advocates suggest that their notations be used. One programming language was dubbed the specification language; the other identified as an implementation language. A program was written by one programmer in the specification language and given to another as a "specification" for a program to be written in the implementation language. The translator often found, and corrected, errors. The "implementation was reviewed by the programmer who had written the specification. The error rate was measurably reduced; the technique (an early version of pair programming) was considered successful. This "dual scrutiny" effect, could explain many of the reported successful applications of formal methods.

Reports that formal methods are ready for industrial use must be taken with a huge grain of salt. The need for effective methods is so great that, if the well-known methods were ready, their use would be widespread.

5 Formal methods and engineering mathematics - contrasts

Noting the widespread use of mathematics in traditional Engineering, and the failure of software developers to use the "formal methods" that were developed to help them, we discuss the differences between formal methods and applied mathematics.

On many occasions, when I have remarked that I advocate the use of mathematics in software development, but oppose "formal methods", I have sensed reactions ranging from puzzlement to amusement or annoyance. The distinction is not obvious to many in the field.

This section presents some differences between mathematics and formal methods and explains their importance.

5.1 Mathematics

Standard dictionary definitions of mathematics such as, "*the abstract science of number, quantity, and space*" tell us more about the origin of mathematics than about mathematics as it is understood today. Mathematics is no longer restricted to "number, quantity, and space";it has been extended to include strings, physical structures, and many other types of objects in our world. Abstract structures defined long ago have proven to be useful in understanding things that were devised quite recently.

Mathematicians work by defining abstract structures and then studying their properties. The structures may be very basic and general such as sets, and relations or more complex, algebraic structures comprising several sets and functions mapping between them. A structure is defined by describing properties of the set members and the functions. Mathematicians may derive further properties from the properties given in the definitions.

Broad classes of structures may be divided into smaller classes by stating additional properties. For example, we may define the broad class *relations* and then define the additional properties of functions and various narrower classes such as functions that can be represented by polynomial expressions, boolean functions, or trigonometric functions.

Mathematical structures are:

- **Abstract**: Mathematical definitions do not refer to the real-world objects or structures that inspired them. The properties are all formally stated without any mention of observed behaviour in the real world.

- **Static:** Mathematical structures do not change with time. Some statements may appear to be describing change. For example, we may illustrate set union with examples like {2,3} ∪ {3,4} = {2,3,4}. This is sometimes be paraphrased as "if you add the set {3,4} to the set {2,3} the result is the set {2,3,4}." Such a sentence

sounds as if it is describing an event or action but the sets are all static; the equation states a permanent property of set union.

- **Precisely defined**: Classical mathematical definitions are mature (well structured, well understood, widely accepted). All conclusions about the structures have been derived from the definitions.

We shall refer to these three properties as the ASP properties. These three properties are very important.

- **Abstract** definition makes it possible to apply research results to many different types of systems. For example results from research motivated by the study of systems of springs and weights, may be applied to electrical circuits or control systems without any change to the mathematics. The same equations arise; the same solution techniques work.

- **Static** structures are easier to study. The idea of time in a dynamic system is often confusing in a way that static structures are not. The fact that the structures are static does not mean that we cannot use mathematics to analyze and design dynamic systems. Many of the most useful mathematical results are about dynamic systems. These results treat time in the same way that they treat other variables.

- **Precise Definitions** are essential for productive discussions of complex products. Without them discussions will be full of misunderstandings and the meaning of agreements will be disputed.

5.2 Formal Methods

The phrase "formal methods" was introduced by computer science researchers to describe a class of approaches to studying the properties of computer programs. The two best known are Z and VDM, but there are many variations of these as well as many other methods that come under this rubric. Among these are B (developed by the originator of Z), SOFL, LOTOS, Larch, SDL, TLA, CCS, SDL, …, . Some would include UML as a formal method but many note that it is not formally defined. A list of all the, notations, the variations and their associated methods is beyond the scope of this paper.

Formal methods comprise (1) a notation (often called a language) for describing or specifying computer systems and (2) procedures for using the language to study a computer system.

Another set of methods introduced for analyzing software introduce little or no new notation. Instead, they show how to convert hypotheses about computer system to theorems which can then be proven using standard mathematical techniques. Paradigmatic of this class of approaches was the pioneering work of Robert

Floyd [6] and his students. Other such approaches will be discussed below.

In this paper, I use the term "formal methods" for methods that are associated with a language or special notation for software. The others will be called "mathematical methods".

5.3 Are "formal methods" and mathematics the same?

The languages used in formal methods resemble mathematics. Many of the symbols used are symbols used in classical mathematics and the formulae have similar interpretations.

The similarities between formal methods and classical mathematics are clear when one reads tutorials or books explaining those languages. The introductions almost invariably begin by presenting basic ideas about set theory and logic (often, as if they were new) and introduce notation that is a syntactic variation of the notations used in mathematics. Concepts like set union, conjunction, implication, etc. are explained using the notation of the method's language.

However, the well-known formal methods do not have the ASP properties.

- Formal Method notations were developed for the specific purpose of describing the behaviour of computer programs. In many of the published examples there is a frequent domain change between the physical situation (the program) and mathematical symbol manipulation. The notions of state, programming language variables[1], and actions (state changes) are basic concepts in the formal method languages. The notations are chosen to make the description of programs more direct. Many of the "specifications" are actually descriptions of abstract state machines whose behaviour mimics that of a program that would satisfy the specification.

- The notion of time as "something special" is built into these languages and methods as they talk about values before and after operations (which change states). In some cases, the languages incorporate specialized logics known as temporal logics. The structures described are not static.

- Many of the languages were introduced without a full formal definition; in some cases there have been efforts to correct the lack of a precise abstract definition long after the language was introduced. Variant definitions are frequently introduced in the literature.

The developers of these methods deviated from classical mathematics in this way because they

---

[1] These are different from mathematical variables - see below.

believed that this was necessary to make their ideas practical.

5.4 Modeling

Recent years have seen a renewed interest in the use of models and the coining of the phrase, "model-driven engineering."

A *model* of a product is a simplified version of that product. There are two kinds of models: physical and abstract.

Although, the popular formal methods are often described as specification languages, it would be more accurate to call them modelling languages. Many of the published examples are neither specifications nor descriptions but models.

The properties of a model are never the same as the properties of the actual system. Consequently, models must be prepared and used with great care. Decisions based on models can be wrong if they are derived from properties of the model that are not properties of the actual product.

6   Abstractions and their representation

Mathematics is the study of abstract structures. The characteristics of these structures are described by axioms or equations. The structure cannot be seen; we can only see examples.

To be useful, a structure (e.g. a function) must be described. A mathematical expression (classically, a string) that that describes the structure is called a *representation* of the object. Thus, when we write

$$\text{“f(x)= x+1”,}$$

the string in quotes is not the function called f but a representation of that function. A given abstract object can have many possible representations. For example,

$$\text{“f(x) = 1+x”, and}$$

$$\text{“f(z) = 2×z -z +1”}$$

are different representations of the same function. Much of mathematics deals finding simpler representation of functions or determining whether two expressions represent the same function.

In natural language discussions of mathematics and its applications, there is an unfortunate tendency to confuse the abstraction with one of its representations. For example, we often see "predicate" referring to an expression that describes a predicate.

Some classes of abstract objects are defined by whether or not they can be represented in a certain form. For example, both the class of <u>expressions</u> of the form:

$$f(z) = a_0 + a_1 \times z + a_2 \times z^2 + \dots a_n \times z^n$$

and the class of <u>functions</u> that can be represented (exactly) by expressions in that form are sometimes called polynomials. The function represented by

$$f(z) = (z+1) \times z$$

is a polynomial function even though the expression is not a polynomial, because that function can also be represented by the expression

$$f(z) = z^2 + z,$$

which is a polynomial expression.

Finding the best representation for performing a mathematical task is an essential step in applying mathematical results. For example, finding the roots of the function described above is (slightly) easier using the first representation than using the polynomial expression.

The issue of representations becomes important to mathematics researchers whenever new applications require them to work with a new class of structures. The use of mathematics to describe the behaviour of software requires us to represent functions with many points of discontinuity. Classical engineering deals primarily with continuous functions or piecewise-continuous functions with a small number of discontinuities. Section 10 discusses the representation of software with many discontinuities.

7   How mathematics is applied to physical systems

To apply mathematical results we have to find a way to connect the abstraction to the physical system that we want to study. This is usually done by using variables. The word "variable" is used in both mathematics and physics, but the meaning is different.

7.1 Variables in physics

In physics, "variable" refers to a measurable characteristic of a system whose value changes with time. To define a variable one must describe a way that it can be measured and the units that will be used. This allows a scientist to make statements about the value of this variable. Usually a variable is identified by a short character string such as "h" or "height". Often the name used to identify the variable is not distinguished from the variable itself.

7.2 Variables in mathematics

In pure mathematics, variables are place holders used in the definition of relations. When used for this purpose, they do not have values. The variables have no meaning outside of the definition. For example, we can write

$$\text{“doublesum(x,y) = x+y+ x+y” .}$$

If we write,

$$\text{“doublesum (w,z) = w+z+ w+z” ,}$$

we have defined exactly the same function. The variables are just a convenient way of saying, "the first

argument", "the second argument", etc. As in physics, a variable is usually represented by a short string.

Mathematical variables are used again when the function is applied. If we want to apply the function to specific values, we may write

doublesum(x,z) for x=2 and z = 3, or
doublesum(2,3).

A list of specific values for the arguments such as (2,3) is often called an assignment of values for (x,z) but mathematical variables do not have values outside of the expression, i.e. the "assignment" has no lasting effect on the .variable.

In informal discussions, there is a natural tendency to confuse the variable with the set of values that we intend to assign to it. For example, taken literally, the statement, "x is an integer", where x is a variable is actually nonsense. A constant such as "2" is an integer but x is a variable not a number. Usually such a statement is intended say that the variable will only be assigned integer values or as a part of an informal conditional statement.

7.3 Associating physical variables with mathematical variables

When mathematics is used to study or describe of physical systems, relations are applied by associating the mathematical variables with physical variables. Usually this is done by a kind of pun, i.e by giving the mathematical variable and the associated physical variable the same name. For example we may write

doublesum(length, width)

where length and width are the names of defined physical variables.

When this punning is used, it may convey the (false) impression that the mathematical variables are the same as the physical variables and some practitioners ignore the distinction. In mathematical reasoning any association between the physical variable and the mathematical variable is irrelevant and should not be used.

7.1 Time as a variable

Time receives no special treatment in classical applied mathematics. In Physics, and all branches of Engineering that deal with dynamic systems, time is treated in the same way as other physical variables.[2] This allows us to predict the behaviour of moving objects, electrical circuits, control systems, etc. In Engineering and Physics there is no need for temporal

logic to describe and analyze the dynamic systems that are ubiquitous in these fields.

7.2 Transfer functions

In traditional Engineering, a device's behaviour is characterized by a function with a domain containing the possible histories of the observable behaviour (usually, the input history suffices) and a range that contains the value of the device's outputs. The history contains information assumed to be known or controllable and the range represents the information we want to know. Transfer functions can describe the behaviour of the device without providing any information about its construction.

Given a network of devices, each described by its transfer function, it is possible to derive the transfer function for the network. This supports a hierarchical documentation and analysis process in which networks can be encapsulated and their transfer function (which is usually simpler than the full network description) used in analysis of the networks of which it is a part. [14]

Analysis allows the detection of anomalous behaviour, such as resonant frequencies or other types of behaviour, that might not be revealed by testing.

8   Mathematical reasoning:

Three distinct forms of reasoning can be observed in the development and application of mathematics.
- function evaluation
- predicate satisfaction
- Inference or deduction

8.1 Function evaluation

Function evaluation begins with an expression that represents a function whose domain is the set of possible values of the variables appearing in the expression and range is the set of possible values of the function. Evaluation proceeds by substituting assigned values for the variables, and evaluating functions when the value of all of its arguments have been computed. The order of evaluation is not fully determined but, when there is a choice, the result is not affected by the order chosen.[3]

8.2 Predicate satisfaction

Predicate satisfaction begins with an expression that represents a function with range {***true***, ***false***}. However, no values are assigned to the variables. Instead, there is a search for an assignment that the function maps to ***true***. Theoretically, the process could be a random search but research has found faster methods. There may be many assignments that would

---

[2] One can consider time to be he fourth dimension of a 4 dimensional space with the first three coordinates taken from a conventional spacial coordinate system.

[3] Readers are reminded that we are discussing mathematical expressions, not programs. There are no side-effects.

map to **_true_**; the order in which assignments are tried can affect the result. In traditional engineering the most common form of the predicates is a set of equations. Predicate satisfaction is then called solving equations.

8.3 Inference or deduction

Logic provides a sound basis for mathematics. A particular logic comprises a set of axioms (predicates assumed to be true) and rules of inference (rules that can be used to transform a predicate expression to another expression). Anything inferred in this way from the axioms is considered to be true. If an expression is hypothesized to be a true, proof procedures search for a sequence of inferences that will demonstrate its truth.

8.4 Mathematical reasoning in Engineering.

Of the three forms of mathematical reasoning outlined above, Engineers prefer the first. If they are given equations, they will try to "solve" the equations to produce what is often called a "*closed form solution*", which is a function that maps from a domain consisting of the possible values of the known (independent) variables, to a range comprising the values of the variables that are not known but needed.

The reason for preferring function evaluation is that inference and predicate satisfaction generally involve a search. In the case of inference it is a search for the right sequence of inference applications. In the case of predicate satisfaction it is a search for the right assignment. In function evaluation there is no need for a search. One is given the values of the independent variables and evaluates functions in an obvious order.

Many Engineering mathematics programmes include little or no discussion of inference. Much of the mathematics curriculum is devoted to solving equations to find expressions that can be evaluated. Previously found solutions are taught in the engineering courses.

Function evaluation is the primary way that mathematics is used in practical engineering applications. Mathematicians use inference or deduction to develop new mathematical results and proof techniques. Predicate satisfaction is used for problems where no "closed form" solution is known.

9   Applying mathematics to software systems

This section discusses applying the methods used in older areas of Engineering in software development.

9.1 Variables and identifiers in programs

Above, I have pointed out that the meaning of "variable" in mathematics is different from its meaning in physics. In discussing programs, the word has yet another meaning.

Variables in programs are discrete state machines that serve as memory. They are generally finite state machines; the upper bound to the number of states may be very large and, for some implementations, hard to characterize. Unlike variables in mathematics, program variables have a value, their state, which can persist over time. They are not placeholders used for definition and application of functions. They are closer to physics variables than the mathematical ones because they correspond to a physical device and the values are observable characteristics of that device [4].

Because of the nature of many programming languages, it is important to distinguish between a variable and a string that is used to identify it (an identifier). In modern programming languages, it is possible for a variable to have several identifiers (aliasing) or for one identifier to be used to identify different variables in different parts of a program.

Because the relation between identifiers and variables is not 1:1, the "punning" method of associating mathematical variables with the program variables can cause difficulties. It is easy to replace identifiers that have been used more than once with unique names for the variables, but aliasing is not as easily avoided. The methods available to deal with aliasing depend on the design of the language. Argument passing in procedures is the main source of the aliasing problem as one can use one variable as an actual parameter corresponding to several distinct formal parameters. Unless one consistently deals with an array as a single variable, array indexing is also a source of aliasing. Another form of aliasing is caused by the use of pointer variables.

The sequel assumes that this problem has been solved in some way.

9.2 Assertion based methods

The best known, and most widely taught, methods of relating software to mathematics is the approach introduced by Robert Floyd, [[6]]. Predicates are associated with points in the program text; predicate transformation rules are given for each type of statement in the language. Starting with a predicate that is assumed to be true before the program is executed, and following each path, one assumes that the assertion before a statement holds and must prove that the assertion after that statement will hold after execution. Demonstrating the correctness of a program is, thereby, reduced to proving a set of mathematical theorems.

Looking at proofs using this basic method one can make the following observations:

---

[4] Algol 60 introduced the distinction between variables (which were declared) and formal parameters (which were specified). Formal parameters were similar to the variables used in mathematics.

- In the proof examples, it is common to move back and forth between the program domain (where one exams the paths in an intuitive way) and a mathematical domain (where theorems are proven). It is possible to separate the two types of work by generating all proof obligations from the program text before moving into the mathematics domain. This may require backtracking if one did not introduce assertions that are strong enough.

- Abstraction is not supported. As one moves through the program text, one carries all state variables in the expressions so that some are mentioned in expressions even where they are not relevant.

- If code is reused, proofs may be repeated.

- Although it is natural to go forward, there is no requirement to do so and one could also go backward.

- Although it is common to regard the precondition and postcondition as separate, it was often necessary to add variables that had no other purpose than to support the proof.

- The process neither assumes nor exploits any hierarchical structure in the text.

Floyd did not introduce a new notation for his method; none was needed. In a slightly later paper, Hoare introduced what has become known as the "Hoare triple", that comprised a precondition pattern, a statement pattern, and a postcondition pattern. The triple was a "schema" describing the way that a predicate expression describing the precondition would be transformed to a predicate expression describing the postcondition. The rules given are valid only if certain assumptions about the variables hold. In particular, the relation between variables and identifiers must be 1:1 and that program variable identifiers cannot appear as bound variables in the predicate expressions.

9.3　Predicate transformer methods

Dijkstra [4] replaced the Hoare triple with a function from predicate expressions to predicate expressions, which he called a *predicate transformer.* In addition, he decided to reverse the direction, i.e. his predicate transformers transformed a postcondition to a precondition. He also took the problems of non-determinism into account and identified two distinct predicate transformers, known as the *weakest precondition* and *weakest liberal precondition.* For a deterministic program, these would be the same but for a non-deterministic program they could differ. Predicate transformers, like the earlier methods, did not abstract from the representation of the state. They were always expressed in terms of a specific form of predicate expressions; Dijkstra also assumes a 1:1 mapping between program variables and mathematical variables.

9.4　Abstract relational/functional methods

Many mathematicians, among the best known were Mills [15] and de Bruijn [2], pointed out that new concepts were not needed because the classical mathematical concept of relation[5] could be used A deterministic program computes a function from starting state to stopping state. For non-deterministic programs, the mapping may be a relation that is not a function. In the non-deterministic case, it is either necessary to introduce a way to denote non-termination or use the approach like that in [16].

Expressing the mathematical properties of programs with functions on states has a number of advantages:

- There is no need for new notation. This is not new mathematics; it is simply a new application for old mathematics.

- The definitions given in papers like [15,16] are independent of the way that the states and functions are represented.

- The definitions are simpler and, as a consequence, easier to understand. The simplicity is the result of abstracting from representation details.

- There is no need for a special treatment of time. Time is part of the state information and the passage of time during the execution can be treated in exactly the same way as changes of other variables. If the execution time is difficult to predict, time consumption can be considered part of the non-determinism.

- The definitions given in papers like [15] are not tied to a specific set of statement types. They distinguish two components of a programming language, primitive (built in "statements" or programs) and constructors (such as "**;**" "**if then else**", and "**while**") which are used to construct bigger programs from primitive programs and previously constructed programs. This eases their application to new programming languages.

- The absence of assumptions about the primitive programs supports hierarchical application of the laws. Any constructed program can be treated as a primitive program when constructing larger ones.

Relational methods are the closest of the software methods to the methods used in classical Engineering. In the author's opinion, they have many advantages. For example, no changes are required to use the tabular notations defined in [13]. The mathematics is applicable to a wide variety of programming languages including those with unusual data types. Using this approach facilitates a smooth integration of Software Engineering and traditional Engineering.

---

[5] Mills work [15] was restricted to deterministic programs and relations that were functions.

9.5 Additional views of software

Early work on the application of mathematics to programs only considered individual sequential deterministic, terminating programs. This was adequate for the programs of concern in the 1960s but it is not adequate for the type of software we build today. For today's systems, we need to view software from many "orthogonal" viewpoints. Each view can be described in a separate document as summarized in Figure 1.

Document	Content
Software Requirements Document	Black Box specification, identifies all outputs and inputs and describes relation between output values and input history.
Module Guide	Informal description showing the hierarchical decomposition of the system into modules and the secret of each module.
Module Interface Specifications	Black Box specification, identifies all outputs and inputs and describes relation between output values and input history." Usually shows externally invokable programs.
Module Implementation Design Document	Documents complete data structure, effects of all externally visible programs on data, abstraction function or relation (data interpretation) - design can be verified before coding begins.
Program Uses Structure	Description of permitted usage of one access-program by another. Determines the usable subsets of product.
Display Method Program Documentation	Hierarchical decomposition of program into "small programs" with specification of subject program and programs used.

Figure 1: Software Design Views

9.6 Applying mathematics for the other views

None of the well-known formal methods was designed with views other than individual terminating programs in mind. Because each method was primarily a language there have been a few attempts to apply the notations to other views but the result is not convincing. In fact, the communities behind these methods do not seem to understand the need for separation of concerns and a multi-view approach.

The relational approach, can be used for each of the views. Each of the views can be documented using mathematical relations but the range and the domain of the relations varies with the view. The contents of each document other than the module guide, which is the only informal document, will be a representation of a relation. [17]. Figure 2 summarizes the relations to be represented in each of the documents.

Document	Domain	Range	Relations
Software Requirements Document	Histories of I/O	Output values	One per output variable: possible value after history
Module Interface Specifications	I/O traces (sequences)	Output values	One per output variable possible value after trace
Module Implementation Design Document Domain	Declaration of complete data structure		
Module Implementation Design Document part 2	Data Structure States	set of traces	Abstraction Relation: Trace could lead to state
Module Implementation Design Document 3	Access Program Function	Data Structure States	Data Structure States
Program Uses Structure	Access Programs	Access Programs	Allowed to Use
Display Method Program Documentation	program start states	program end states.	program functions (with text)

Figure 2: Relational Content of Primary Software Design Documents

With a multiple view approach to software, it becomes extremely important to have clear statements of the contents of each document. Many developers have witnessed destructive arguments about whether some detail should be included in a given document or belongs elsewhere. More have witnessed misunderstandings because the information was discussed in more than one document and the documents were not consistent. Mathematics gives us a way to say precisely what belongs in each document without specifying format or notation [16].

It is also possible to use mathematics to verify key design decisions before spending time and money on implementation of faulty interfaces. Methods used in other areas of Engineering can be used to check higher-level documents before investing in code. [14]

10 Representing Software Relations

The mathematical methods that have been used in traditional Engineering fields have not been easy to apply to software because the classical representations are not suitable for the functions that must be described. While the functions that arise when working with physical products are usually either continuous or have a small number of discontinuities, the functions that describe software have many discontinuities, both because we are dealing with digital systems and because the power of software lies partly in its ability to implement behaviour that has many special cases.

Mathematical expressions that describe computer systems can become very complex, hard to write and hard to read. When software functions are described by expressions in conventional format, the depth of nesting of subexpressions gets to high. As first demonstrated more than 30 years ago in [8, 7], the use of a tabular format for mathematical expressions can turn an unreadable string of symbols into an easy to access, complete and unambiguous document.

Figure 3 [6] is an expression that describes the behaviour of a keyboard checking program that was developed by Dell in Limerick, Ireland [1, 18]. Even those who are mathematically inclined find such expressions hard to read and write.

Keyboard Checker: Conventional Expression

(N(T)=2∧keyOK∧(¬(T=_)∧N(p(T))=1))∨(N(T)=1∧(T=_∨(¬(T=_)∧N(p(T))=1))∧
(¬keyOK∧¬prevkeyOK∧¬prevkeyesc))∨((¬(T=_)∧N(p(T))=1)∧
((¬keyOK∧keyesc∧¬prevkeyesc)∨(¬keyOK∧keyesc∧prevkeyesc∧
prevexpkeyesc))∨((N(T)=N(p(T))+1)∧(¬(T=_)∧(1<N(p(T))<L)∧(keyOK))∨
((N(T)=N(p(T))-1))∧(¬keyOK∧¬keyesc∧(¬prevkeyOK∧prevkeyesc∧
preprevkeyOK)∨prevkeyOK)∧((¬(T=_)∧(1<N(p(T))<L))∨(¬(T=_)∧N(p(T))=L)))∨
((N(T)=N(p(T)))∧(¬(T=_)∧(1<N(p(T))≤L))∧((¬keyOK∧¬keyesc∧(¬prevkeyOK∧
prevkeyesc∧preprevkeyOK))∨(¬keyOK ∧¬prevkeyOK∧ ¬prevkeyesc)∨
(¬keyOK∧keyesc∧¬prevkeyesc)∨(¬keyOK∧keyesc∧prevkeyesc∧
prevexpkeyesc))∨((N(P(T)=Fail)∧(¬keyOK∧keyesc∧prevkeyesc∧
¬prevexpkeyesc)∧(1≤N(p(T))≤L))∨(N(P(T)=Pass)∧(¬(T= _)∧N(p(T))=L)∧(keyOK))

Figure 3: Characteristic Predicate of Keyboard Checker Program

---

[6] Auxiliary predicates such as keyesc, keyOK, etc. are defined separately. Each is simply defined.

Figure 4 is a tabular mathematical expression describing the same function. It comprises 3 elements called grids. Each grid contains a number of cells. For this type of expression, the top grid and the left grid are called "headers". The lower right grid is the "main grid".

To use this type of tabular expression, one evaluates the cells the headers to find the ones that evaluate to ***true***. If the table has been properly constructed, exactly one of the column header cells and one of the row header cells will evaluate to ***true*** for any assignment of values to the variables. The indices of those cells identify a row and column. Together they identify one cell in the main grid. Evaluating the expression in that cell will yield the value of the function.

A tabular expression parses the conventional expression for the user. Instead of trying to parse, evaluate or understand the complex expression in figure 3, one looks at the simpler expressions that appear in the tabular expression. Generally, one will only need to evaluate a few of the expressions that appear in the cells to compute an answer. Note that although the tables use predicate expressions, the user is evaluation functions, not using predicate satisfaction or inference.

There are many forms of tabular expressions. The grids need not be rectangular. The format is not limited to two-dimensional grids or tables. Any expression in any cell can itself be a tabular expression. A variety of types of tabular expressions are defined in [13].

Keyboard Checker: Tabular Expression

N(T) =

			$\neg (T = \_) \wedge$		
		T=_	N(p(T))=1	1<N(p(T))< L	N(p(T))= L
keyOK			2	N(p(T))+ 1	Pass
¬keyOK ∧	¬keyesc ∧	(¬prevkeyOK ∧ prevkeyesc ∧ preprevkeyOK) ∨ prevkeyOK		N(p(T)) - 1	N(p(T)) - 1
		¬prevkeyOK ∧ prevkeyesc ∧ ¬preprevkeyOK		N(p(T))	N(p(T))
		¬prevkeyOK ∧ ¬ prevkeyesc	1	1	N(p(T))	N(p(T))
	keyesc ∧	¬ prevkeyesc		1	N(p(T))	N(p(T))
		prevkeyesc ∧ ¬prevexpkeyesc		Fail	Fail	Fail
		prevkeyesc ∧ prevexpkeyesc		1	N(p(T))	N(p(T))

Figure 4: Tabular Expression for Figure 1

Although tabular expressions were successfully used without proper definition for many years, precise semantics are needed for tools and full analysis. There have been four basic approaches to defining the meaning of these expressions. Janicki and his co-authors developed an approach based on information flow graphs that can be used to define a number of expressions.[11]. Zucker based his definition on predicate logic.[19] . Khédri and his colleagues based their approach on relational algebra [3]. All three of these approaches were limited to basic forms of tables [12]. The most recent approach, [13], is less restricted; it defines the meaning of these expressions by means of translation schema that can be used convert any

tabular expression of a known type to an equivalent conventional expression. This is the most general approach and provides a good basis for tools. The appropriate table form will depend on the characteristics of the function being described. Jin shows a broad variety of useful table types and which provides a general approach to defining the meaning of any new type of table [13].

This newer approach to tabular expressions makes it possible to chose a function representation that is intuitive and easy for its intended audience to use without losing the precision of mathematics. Both commercial and academic prototype tools have demonstrated the ability to use this type of representation to check information for completeness and consistency as well as to check correctness by automatic construction of a prototype that can be used in testing.

Perhaps the most important thing to note about tabular methods in software development is that tabular expressions are "closed form" expressions and can be used for function evaluation; this is easier than using the inference approach, which is the most common approach used in formal methods.

11 Conclusions

Software developers and educators must ask a simple question, "Why is mathematics, which has often been shown to be an essential tool in traditional engineering, so seldom used in software development? Computer Scientists have made three basic mistakes.

• They based their "formal methods" on the way that mathematicians and philosophers develop new mathematics, rather than the way that Engineers have applied mathematics to design products.

• They failed to recognize that mathematicians always have developed new forms for expressions whenever they study a new class of functions. The functions that describe software behaviour are quite different from those encountered when describing physical products; new notation was needed.

• They failed to think deeply about the roles that mathematics could play in software development, i.e in specifications, in descriptions, and in verification. For the most part, they thought only about verification and failed to distinguish between, modelling, description and specification.

We now have a clearer picture of the ways that mathematics can and should be used, and a much better notation for describing functions and relations. It is time for software developers to act more like traditional Engineers and make use of the powerful mathematical tools available to them.

12 References

[1] Baber, R., Parnas, D.L., Vilkomir, S., Harrison, P., O'Connor, T., "*Disciplined Methods of Software Specifications: A Case Study*", Proceedings of the International Conference on Information Technology Coding and Computing (ITCC 2005), April 4-6, 2005, Las Vegas, NV, USA, IEEE Computer Society.

[2] de Bruijn, N. G., "*The Mathematical Language AUTOMATH – Its Usage And Some Of Its Extensions*" In Symposium on Automatic Demonstration, volume 125 of Lecture Notes in Mathematics, pages 29–61. Springer-Verlag, 1970.

[3] Desharnais, Jules, Khédri, Ridha; Mili Ali, "*Towards a Uniform Relational Semantics for Tabular Expressions*" *Proceedings of RelMiCS* 1998, pp. 53-57

[4] Dijkstra, E.W., "*A Discipline of Programming*", Prentice Hall, Englewood Cliffs, NJ, 1976

[5] Elovitz, H. S., "An experiment in software engineering: *The Architecture Research Facility as a case study*", Proceedings of the 4th International Conference on Software engineering Munich, Germany, Pages: 145 - 152, 1979

[6] Floyd, R.W., "*Assigning Meanings to Programs*", Proceedings of the Symposium of Applied Mathematics, Vol.19, 1968. Also in: Schwartz, J.T. (editor), Mathematical Aspects of Computer Science, American Mathematical Society, pp. 19-32, 1967.

[7] Heninger, K., Kallander, J., Parnas, D.L., Shore, J., "*Software Requirements for the A-7E Aircraf*t", Naval Research Laboratory Report 3876,Nov. 1978, 523 pgs.

[8] Heninger, K.L., "*Specifying Software Requirements for Complex Systems: New Techniques and their Application*", IEEE Transactions Software Engineering, Vol. SE-6, pp. 2-13, January 1980.

- Reprinted as chapter 6 in [10]

[9] Hester, S.D., Parnas, D.L., Utter, D.F., "*Using Documentation as a Software Design Medium*", Bell System Technical Journal,60,8, pp.1941-1977, October 1981

[10] Hoffman,D.M.,Weiss, D.M. (eds.), "*Software Fundamentals: Collected Papers by David L. Parnas*", Addison-Wesley, 664 pgs., ISBN 0-201-70369-6,

[11] Janicki, R. "T*owards a Formal Semantics of Parnas Tables*", Proc. of 17th International Conference on Software Engineering, (ICSE), pp. 231-240 1995,.

[12] Janicki, R., Parnas, D.L., Zucker, J., "*Tabular Representations in Relational Documents*", in "Relational Methods in Computer Science", Chapter 12, Ed. C. Brink and G. Schmidt. Springer Verlag, pp. 184 - 196, 1997, ISBN 3-211-82971-7.

- Reprinted as Chapter 4 in item [10].

[13] Jin, Ying, Parnas, D.L., "*Defining The Meaning Of Tabular Mathematical Expressions*", Science of Computer Programming (Elsevier), Vol. 75, Issue 11, 1, Pp. 980-1000, doi:10.1016/j.scico.2009.12.009, November 2010

[14] Liu, Zhiying , Parnas, D, L, Trancón y Widemann, B., "*Documenting and Verifying Systems Assembled from Components, Frontiers of Computer Science in China*", Higher Education Press, co-published with Springer-Verlag GmbH ISSN1673-7350 (Print) 1673-7466 (Online), 2010.

[15] Mills, Harlan D.: "*The New Math of Computer Programming*". *Comm. ACM 18,*1): 43-48 (1975)

[16] Parnas, D.L., "A *Generalized Control Structure and its Formal Definition*", Comm. ACM, 26, 8, pp. 572-581, Aug. 1983

[17] Parnas, D.L.,Madey, J., "*Functional Documentation for Computer Systems Engineering*", Science of Computer Programming (Elsevier) vol. 25, no. 1, pp 41-61, Oct. 1995

[18] Parnas, D.L., Vilkomir, S.A,. "*Precise Documentation of Critical Software*", Proc. of the Tenth IEEE Symposium on High Assurance Systems Engineering (HASE), 14-16 pp. 237-244, Nov. 2007

[19] Zucker, J.I. "*Transformations of Normal and Inverted Function Tables*", Formal Aspects of Computing 8, pp. 679-705,1996.