Precise Description and Specification of Programs

David Lorge Parnas, P.Eng.
NSERC/Bell Industrial Research Chair in Software Engineering
Director of the Software Engineering Programme
DEPARTMENT OF COMPUTING AND SOFTWARE
Faculty of Engineering
McMaster University
Hamilton ON Canada L8S 4K1

Abstract

Precise descriptions and specifications of programs can be very useful if they are simpler than the products that they describe. No new mathematical concepts are needed for this task; we need only use old math in relatively new ways.

We discuss the difference between descriptions of programs, specifications of programs, and models of programs

What do we want to do with mathematical descriptions of software?

(1) Describing software products that we already have - so that people can use them without reading the code.

(2) Writing specifications for software products we do not yet have - so that the programmers and clients can agree on the requirements.

(3) Be able to verify that a product meets its requirements (testing or proving).

What are the Criteria?

(1) Descriptions must be easier to read than the code.

(2) We must state the requirements in a way that does not restrict the solutions unnecessarily.

(3) Testing and proof can eventually be automated.

Are Programs Different From Other Engineering Products?

Before we had computers, engineers used classical mathematics to describe and analyse their products.

In Computer Science, most researchers have turned to newly invented "languages".

We are using software to replace conventional products.

Why can't we simply go on using the mathematics we used to use?

Right Answer: The functions have many more points of discontinuity. We will return to this point later.

Wrong Answer: Conventional products are inanimate objects.

Wrong Answer: We need to describe the procedure followed by the program.

Describing Engineering Products

Engineers use mathematics, not just words, to describe their products.

- They use a variety of descriptions rather than attempt one "complete" description.
- There is never a complete description of a product.
- Each product description is intended for a different purpose and each is an accurate description of some aspect of the product.
- Even taken together, these descriptions never constitute a complete description. There are always some facts that are not stated in the descriptions.

Even for simple physical objects, Engineers produce several drawings ("views") and require additional numerical specification sheets

Current specification "languages" make no provision for this. They use the same approach for all "views"!

Types of Program Descriptions

In this talk, I distinguish 3 types of descriptions:

- *constructive descriptions*, which show how a program is composed of other programs,
- *behavioural descriptions*, which describe the *visible* behaviour of a program without discussing how it was constructed, and
- *specifications*, which describe requirements that a program must meet.

Texts in a "programming language" are constructive descriptions. Each "language" contains

- constructors, and
- primitive programs.

The other two are the main subject of this talk.

Descriptions vs. Specifications

Engineers make a distinction between specifications of products and other descriptions of those products.

Definition 1:

An actual description is a statement of some *actual* attributes of a product, or set of products. ❑

Definition 2:

A specification is a statement of all properties *required* of a product, or a set of products. ❑

In the sequel, "*description*", without modifier, means "actual description".

- A description may include attributes that are not required.
- A specification may include attributes that a (faulty) product does not possess.
- The statement that a product satisfies a given specification constitutes a description.

The third fact results in much confusion. A useful distinction has been lost.

Descriptions vs. Specifications

Any list of attributes may be interpreted as *either* a description or a specification.

Example:

"A volume of more than 1 cubic meter"

This could be either an observation about a specific box or, a requirement for a box that is about to be purchased.

A specification may offer a choice of attributes; a description describes the actual attributes, but need not describe them completely.

Sometimes one may use one's knowledge of the world to guess whether a statement is a description or a specification.

Example:

"Milk, badly spoiled"

Guessing is not reliable. We need to label specifications and descriptions to distinguish them.

Programs and Executions

In engineering it is vital to clearly state the relationship between the mathematical terms and the physical objects. We must do that here;

A digital computer will be viewed as a finite[1] state machine and a program as a description of a behaviour pattern for that machine.

Definition 3:

A *finite state machine* is a machine that is always in exactly one of a finite set, S, of stable states and whose operation consists of a sequence of state changes, i.e. transitions from state to state. These machines have a finite set of input symbols, called the *input alphabet*, and a finite set of output symbols, the *output alphabet.* ❑

It would be a mistake to confuse the physical machine with its description by saying "a finite state machine is an n-tuple...".

Definition 4:

An *execution* is a sequence (either finite or infinite) of states. ❑

[1] Much of the "theory" applies if the machine had an infinite number of states, but in applications it is essential to remember the finiteness.

Programs and Executions

It is convenient, and conventional, to represent the states of the machine as having two components (P,s). P, which is usually fixed, is called the program whereas s, called the data state, may vary.

This "structuring" of the state is actually quite arbitrary. Information may be moved between P and s subject only to the condition that P not change while the machine is running.

In the sequel, we will assume such a division and S refers to the set of possible data states for a given P.

Definition 5:

A *program* determines a set of possible executions, sometimes called the executions of that program. The set of all executions of P for a set, S of possible data states, is denoted *Exec(P,S)*. ❑

Definition 6:

The subset of Exec(P,S) that begin with the state (P,x), ((P,x) ∈ S), is denoted by $e_P(x)$, and (P, x), or simply x, is referred to as the starti*ng state* of those executions. ❑

Programs and Executions

Definition 7:

If there exists an execution in $e_P(x)$ that is finite and its last data state is z, we write
$\langle x,\ldots, z\rangle \in e_P(x)$, and say that this *execution terminates (in z)*, and call z the *final state (of this execution)*.
We may also say, "the *program P* may *start in x* and *terminate in z*. ❑

Definition 8:

An infinite sequence in $e_P(x)$ ($\langle x, \ldots\rangle$) is called a *non-terminating execution*. ❑

Programs and Executions

Definition 9:

If there exists a data state x, ((P,x) ∈ S), such that $e_P(x)$ contains two or more distinct executions, then P is called a *non-deterministic program*. ❑

Definition 10:

If for a given data state x, ((P,x) ∈ S), every member of $e_P(x)$ is finite, x is called a *safe state* of P. The set of safe states of P is denoted S_P. ❑

A Mathematical Interlude: LD-Relations

Definition 11:

A *binary relation* R on a given set U is a set of ordered pairs with both elements from U,
i.e. $R \subseteq U \times U$.
The set U is called the *Universe of R*.
The set of pairs R can be described by its *characteristic predicate*, R(p,q),
i.e. R = {(p,q): $U \times U$ | R(p,q) = true}. The *domain* of R is denoted Dom(R) and is {p | ∃q [R(p,q)]}.
The *range* of R is denoted Range(R) and is {q | ∃p [R(p,q)]}.
Below, "relation" means "binary relation". ❑

Definition 12:

A *limited-domain relation* (LD-relation) on a set, U, is a pair, $L = (R_L, C_L)$, where:
R_L, the *relational component* of L, is a relation on U, i.e. $R_L \subseteq U \times U$, and
C_L, the *competence set* of L, is a subset of the domain of R_L, i.e. $C_L \subseteq Dom(R_L)$. ❑

Behavioural Descriptions of Programs

Definition 16:

Behavioural descriptions describe some aspects of the executions of a program; they need not describe how the program is constructed from component programs. ❑

For example, performance models are behavioural descriptions.

Those who are going to use, not inspect or modify, a program need behaviour descriptions far more than they need constructive descriptions of programs.

Even programmers need behavioural descriptions of constructed programs. The constructive descriptions of "large" programs (programs constructed by using many applications of the constructors) are very hard to understand.

Before/After Descriptions

Before/after descriptions are a class of behavioural descriptions that are used when the intermediate states of an execution are not important. For each state, s, they must describe:

- (a) whether or not s is safe, and
- (b) the set of final states of the executions in $e_P(s)$.

The methods used in most approaches to program verification are forms of before/after descriptions.

Using LD-Relations as Before/After Descriptions (1)

Definition 17:

Let P be a program, let S be a set of states, and let $L_P = (R_P, C_P)$ be an LD-relation on S such that $(x,y) \in R_P$ if and only if $\langle x,\ldots,y\rangle \in Exec(P,S)$, and $C_P = S_P$.[1] L_p is called the *LD-relation of P* ❑

By convention, if C_P is not given, it is, by default, $Dom(R_P)$.

[1] Please note that C_P is not the same as the precondition used in VDM [4]. S_P is the safe set of P.

Using LD-Relations as Before/After Descriptions (2)

The following follow from Definition 17:

- If P starts in x and $x \in C_P$, P will always terminate; if $(x, y) \in R_P$, P may terminate in y.
- If P starts in x, and $x \in (Dom(R_P) - C_P)$, the termination of P is non-deterministic; in this case, if $(x, y) \in R_P$, when P is started in x, the execution may terminate in y or may not terminate.
- If P is started in x, and $x \notin Dom(R_P)$, then the execution will not terminate.
- If P is a deterministic program, the relational component, R_P, is Mills' program function and C_P (which will be exactly $Dom(R_P)$) need not be written. Hence, our approach is "upward compatible" with Mills' [2,3].

By these conventions we are able to provide complete before/after descriptions of any program but have an efficient representation for those cases that arise most often.

Using LD-Relations as Before/After Descriptions (3)

LD-relations have advantages over other, more popular, before/after descriptions.

- They provide complete before/after descriptions of non-deterministic programs.
- They can be described by giving the characteristic predicates of the Relation and Competence Set; those predicates can be expressed in terms of values of the actual, program variables.

The last property is of great practical value.

It allows us to use completely conventional (classical) mathematics yet still provide descriptions in terms of things that programmers know about.

Alternative Behavioural Descriptions

(1) Before/after descriptions do not describe invariants. Users of a program do not need to know the invariant of its loops.

(2) VDM does not describe the behaviour of a program if the precondition doesn't hold. The VDM model does not allow one to distinguish certain programs that have distinct before/after behaviour.

(3) Pre/Post conditions are a fiction. We are not really interested in two separate conditions and are often forced to add (otherwise unnecessary) variables so we can describe a relation.

Alternative Behavioural Descriptions

A standard (decades older) alternative uses a single relation with a special symbol (e.g. "⊥") representing non-termination.

- This is mathematically (theoretically) equivalent. You can derive the LD-Relation from it and vice versa.
- ⊥ is not a state and exceptions must be made for it when giving the algebra of relations. Certain statements of algebraic laws become more complex.
- The characteristic predicate of the relation cannot be written in terms of variable values alone. One must add something. This is a practical disadvantage.

The Sets/relations can be characterised by predicates. This gives us "predicative programming" but for the completely general behavioural description you need two predicates or some other artifact.

Specifying Programs

Specifications may allow behaviour not actually exhibited by a satisfactory program.

We can also use LD-relations as before/after specifications:

To do this we must describe how the description of a program should be related to a specification that has been provided.

Definition 18:

Let $L_p = (R_P, C_P)$ be the description of program P.
Let S, called a *specification*, be a set of LD-relations on the same universe and $L_S = (R_S, C_S)$ be an element of S.

We say that:
(1) P *satisfies the LD-relation* L_S, **iff** $C_S \subseteq C_P$ and $R_P \subseteq R_S$, and
(2) P *satisfies the specification* S, **iff** L_p satisfies at least one element of S. ❑

Often, S has only one element. If $S = \{L_S\}$ is a specification, then we can also call L_S a specification.

Specifying Programs

The following follow from Definition 18.

- A program will satisfy it's own description as well as infinitely many other LD-relations.
- An acceptable program must *not* terminate when started in states outside $Dom(R_S)$.
- An acceptable program must terminate when started in states in C_S ($C_S \subseteq Dom(R_P)$).
- An acceptable program may only terminate in states that are in $Range(R_S)$.
- A deterministic program can satisfy a specification that would also be satisfied by a non-deterministic program.

Note the following differences between the description and the specification of a program.

- There is only one LD-relation describing a program, but that program will satisfy many distinct specifications described by different LD-relations.
- An acceptable program need not exhibit all of the behaviours allowed by R_S ($R_P \subseteq R_S$).
- An acceptable program may be certain to terminate in states outside C_S. ($C_S \subseteq C_P$).

The interpretation of LD-relations *must* be stated explicitly!

The "Laws" of Programs

Do Software Engineers have laws for programs that correspond to Kirchoff's laws for circuits?

Is there an equivalent of physics for programs?

Yes!

The basic laws of programs are essentially the axioms of the algebra of relations.

If you accept the fact that LD-relations provide adequate descriptions of program behaviour, sequential execution is relational composition.

The proofs are easy if you use classic results about relations.

These laws allow you to find behavioural descriptions of constructed programs if given:

- the constructive description of those programs and,
- the behavioural descriptions of the primitive programs.

What New Notation *do* we Need?

Although the mathematics is old, and the abstract notation for defining things is old, the applications are new.

We have to describe relations and functions that have non-heterogeneous ranges and domains and can have a discontinuity at arbitrary points.

We have found a variety of tabular notations to be useful.

Ryszard Janicki, has found new ways to unite these tabular notations.

Jeff Zucker, Martin von Mohrenschildt and I and our students are implementing tools for transformations.

We are trying to:

- Make the documentation easier to produce
- Make the documentation more useful

A Simple Conventional Expression

$$(((\exists\ i,\ B[i] = x) \wedge (B[j'] = x) \wedge (present' = \textbf{true})) \vee ((\forall\ i,\ ((1 \leq i \leq N) \Rightarrow B[i] \neq x)) \wedge (present' = \textbf{false}))) \wedge (\text{`x = x'} \wedge \text{`B = B'})$$

A tabular expression:

Specification for a search program

$(\exists\ i,\ B[i] = x)$	$(\forall\ i,\ ((1 \leq i \leq N) \Rightarrow B[i] \neq x))$

j' |	B[j'] = x	*true*
present'=	true	false

$\wedge$ NC(x, B)

The above is one of many kinds of tables!

Simple tables like this *understate* the advantage.

These have "practitioner appeal".

They can be used with any mathematical method.

Conclusions

- The mathematics of program descriptions can be kept quite simple without losing utility. We don't need complex new theories.
- "Minor" changes can remove descriptive power.
- The difference between program description and program specification can be made precise, even though the same mathematical formalism can be used for both. Using these words interchangeably, as is often done, is a mistake.
- Our requirements for descriptions of objects and programs are different; a different approach is needed.
- In many years of participation in practical software development, I have seen many places where "theory" or mathematics was useful, but it was always simple theory.
- When theories get complex, they are not likely to be useful or used.
- Those who have defined new languages have not given enough thought to what they are describing.
- We may need new mathematics, but we have not yet exhausted what the old mathematics can do.

Vector Function Table

	H_1: 'X > 7	'X= 7	'X< 7
X' =	'X + 1	100	'X- 1
	H_2		G

Vector Relation Table

	H_1: 'X > 7	'X = 7	'X < 7
X' |	X' = 'X + 1	(X' = 8) ∨ (X' = 6) ∨ (X' = 100)	X' = 'X-1
	H_2		G

Assume functions MAXIMUM, MINIMUM defined on arrays and subarrays.

H_1: $(1 \le l \le u \le n)$	$\neg(1 \le l \le u \le n)$

Assume functions MAXIMUM, MINIMUM defined on arrays and subarrays.

	$(1 \leq l \leq u \leq n)$	$\neg(1 \leq l \leq u \leq n)$
i' |	X[i']=MINIMUM(X[*l*:u])	***true***
j' |	X[j']=MAXIMUM(X[*l*:u])	***true***

H_2 **G**

$\wedge$ NC (X, n, *l*, u)

$(\exists\, l, (l > \text{`k}) \wedge (\forall\ i,\ 0 \leq i \leq p \Rightarrow \text{pat[i]=dat[}l\text{+i]}))$	$\neg\ (\exists\, l, (l > \text{`k}) \wedge (\forall\ i, 0 \leq i \leq p \Rightarrow \text{pat[i]=dat[}l\text{+i]}))$

H1

k' |	k' =minel({*l* | $(\forall\ i, 0 \leq i \leq p \Rightarrow$ pat[i]=dat[*l*+i]) $\wedge$ (*l* >'k)})	***true***
m'=	**true**	**false**
i' |	i' = p+1	***true***

H2 G