Mathematical Foundations and Notation of RESOLVE

Wayne D. Heym
Timothy J. Long
William F. Ogden
Bruce W. Weide

Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210
{heym,long,ogden,weide}@cis.ohio-state.edu

Technical Report OSU-CISRC-8/94-TR45 (Aug. 1994; slightly revised Oct. 1994, Nov. 1994, Feb. 1996, June 1997, Sept. 1998)

Abstract — The RESOLVE approach to reusable component-based software engineering is mathematically-based. This paper discusses the logical foundations and terminology of RESOLVE, the built-in RESOLVE notation for writing mathematics, and the RESOLVE mechanisms that support description of mathematics that has no built-in notation. It is intended to serve primarily as a reference document.

THIS PAGE INTENTIONALLY BLANK

1. Introduction

RESOLVE is our framework, language, and discipline for supporting component-based software [Ogden 94, Edwards 94, Bucci 94]. Formal mathematical notation plays two important roles in RESOLVE, and consequently must support two apparently conflicting objectives:

(O-1) to communicate to the (human) reader of a specification the precise intent of the specifier; and

(O-2) to permit formal (human and/or mechanical) checking that specifications and implementations have certain properties, e.g., that an implementation is correct with respect to its specification.

The mathematical notation must be easily readable, comprehensible, and unintimidating in order to meet objective O-1, but also unambiguous and firmly grounded in order to meet objective O-2. There are trade-offs involved in simultaneously meeting both. To illustrate, we first review some basic terminology from logic and discuss related RESOLVE-specific terminology. Then we summarize the built-in mathematical notation of RESOLVE, and finally introduce the RESOLVE mechanisms for defining other supporting mathematics.

1.1. Logic Background

You might expect a mathematical notation for formal specification of software to provide the ability to describe mathematical models directly in terms of the usual ground notions of set theory, with everything — including integers, functions, relations, etc., — defined in terms of sets. RESOLVE does this in a slightly roundabout manner because we designed its mathematical notation to meet both objectives O-1 and O-2. Common sense and experience suggest that human understanding and communication, and high-level rigorous (but still officially informal) reasoning, have everything to do with *meaning*. On the other hand, well-established principles of mathematical logic suggest that formal proofs should be based entirely on syntactic manipulation.

The notions of "proof" and "meaning" occupy uniquely important places in the history of mathematics. Predictably, understanding the relevant results requires a little background and terminology. We start with a given (fixed) set of formulas, i.e., well-formed strings of symbols, which are called *axioms*. By applying some prescribed symbol-manipulation rules starting from the axioms you can *prove* other formulas, which are called *theorems*. Because the symbols used in these formulas have no particular meanings, theorems are not necessarily "true" in the usual sense of the word; they are simply formulas that can be obtained from other formulas by approved syntactic juggling. The axioms and all theorems provable from them in this way constitute a *theory*.

"Meaning" comes via an *interpretation* of a theory, i.e., a mapping that assigns a set of values (the *universe of discourse*, as discussed in the companion paper on specifications [Edwards 94]) to every *type* of the theory; and a total function or relation to every *operator* of the theory. For example, here is (a part of) one interpretation for a theory of integers:

- The type *integer* is assigned the set of abstract mathematical values you ordinarily think of as "the integers". In principle, this can be formalized in set theory in various ways, but the usual intuitive meaning of "integer" is what we wish to elicit from the reader.

- The (infix binary function) operator + is assigned the function whose value is the sum of its two arguments. Again, in principle a formalization in set theory is a technical possibility, but the intuitive meaning of the "sum function" is what you should think of.

- The (infix binary predicate) operator = is assigned the binary relation containing exactly those pairs of integers whose components have the same value; or, equivalently, the boolean-valued function which is true exactly when its two arguments are equal.

A constant can be considered a 0-ary operator, i.e., an operator with no parameters. In a use of a 0-ary operator, we usually omit the parentheses that enclose the (empty) argument list. So, for example, continuing the interpretation above:

- The (constant) operator *1* is assigned the 0-ary function whose value is the integer you normally think of as "1".

A useful interpretation of a theory has the property that every theorem is actually true when the symbols have the meanings assigned by that interpretation. An interpretation with this property is called a *model* for the theory. There may be many non-isomorphic models for a theory. Proofs that rely only on symbol-manipulation rules are therefore important, not just because they can be automatically checked and (in some cases) generated, but because a formula that is proved by symbol manipulation is true in every model for the theory — no matter which of these meanings is assigned to its symbols.

In particular, every theorem is true in the model that is being used by the writer of a specification to express intent, and by the reader of that specification to decipher intent, even if their models are not the same. But for psychological reasons and to best achieve objective O-1, both writers and readers of specifications probably should think in terms of a *standard model* for each theory. To facilitate this, RESOLVE's built-in mathematical notation concentrates on a few fundamental and well-understood theories that capture the properties of standard models underlying traditional mathematics. With these standard interpretations, formulas mean what you think they should.

Just as there is no technical reason to limit the models of theories to the standard ones, there is no technical reason to limit the theories you may use to the few that are built-in to RESOLVE's notation. There is a way to define new theories; see Section 3. But we don't recommend using it casually, because defining a theory is not something to be done by amateur mathematicians (say, software engineers). History demonstrates that it's easy to get it wrong, and the consequences can be serious if you expect to specify and prove things about, say, safety-critical software systems.[1]

---

1 Z [Spivey 89] and Larch [Guttag 93] are two formal specification languages used today that help establish some context for RESOLVE. Z leans heavily toward meeting objective O-1, while Larch leans heavily toward meeting objective O-2. Z deals directly with sets, functions, and relations. But there is no connection to any implementation programming language, and efforts in the Z community have concentrated more on "refinements" of specifications than on verification of implementations. Larch, on the other hand, has its roots in algebra and has been used as the basis for program verification. But despite suggestions that you should try to reuse traits (theories), as a specifier using Larch you often write a new trait for a new specification; the best evidence for this is the plethora of Larch traits. Even if you are very experienced and can use the Larch tools to help you write a consistent and useful theory, it has no standard model that you and a reader agree to. So as a reader, you usually have to try to understand a Larch specification by studying some (unfamiliar) axioms. We have positioned RESOLVE as a compromise approach that offers the ability to verify programs with the mathematical formality of symbol manipulation, while retaining the relative comprehensibility and readability of direct modeling.

1.2. RESOLVE Terminology

In some places, RESOLVE's mathematical terminology differs from the traditional.[2] One fact justifies this minor complication: We are writing mathematical assertions for the specific purpose of specifying software. The parallels between programming and mathematics explain why words like "type" appear in both contexts, and we need to control the confusion. In fact, we explicitly establish our programming and mathematical terminology to highlight the many connections between programming and mathematics, thereby giving us slightly non-traditional (to mathematicians) terminology in some cases.

In RESOLVE, a type in a mathematical theory is called a *math type*. An operator is called a *math operation*. The "math" prefix in these terms emphasizes that the ideas are analogous to (but of course not exactly the same as) their programming cousins. We use "operation" in place of "operator" to emphasize subtly that, for ease of understanding, the reader should concentrate on the function in the standard model with which the operator of the theory is associated, and not on the symbol itself.

There is no separate RESOLVE-specific term for a predicate. Instead we treat a predicate as a math operation whose result is of the built-in math type **`boolean`**.

1.3. Math Subtypes

A model for a theory assigns a set of values to each math type and a total function to each math operation. But when specifying program behavior, you often want to constrain a programming type's mathematical model to range over a subset of the universe of discourse for some theory. Similarly, you often care about how a function is defined only on a subset of the universe of discourse. Although technically there is no problem handling these situations, it is easy to confuse the reader of such a specification, especially in the latter case.

For example, consider a math type "rational" and a math operation "reciprocal" (which is not constrained by a theory of rationals for an argument of "0"). Consider a standard model that assigns to "rational" the set S whose elements are what you normally consider to be the rational numbers, and to "reciprocal" a function $F: S \to S$ whose value at x is what you normally consider to be the reciprocal of x. There are infinitely many such models that are identical, except that they disagree on F(0). This means that, say, the assertion "reciprocal(0) = 1" is true in some models and false in others.

Of course, this assertion is not a theorem. The potential confusion arises because neither is "reciprocal(0) ≠ 1", although a casual reader might conclude otherwise by arguing that "reciprocal(0)" is "undefined". It is not undefined, because F is total; its value is simply unknown. It would be nice if you (or a tool) could detect and flag assertions like this that might be misleading.

---

2 Actually, no tradition is well-established here. Some authors call types "sorts", and some call operators "function symbols" and "predicate symbols", among other things.

To this end, we introduce math subtypes.[3] A *math subtype* is an identifier that plays the same syntactic role as a math type, but which is characterized by a *constraint* assertion that defines a subset of the universe of discourse. Each math subtype has a parent math subtype, possibly a grandparent, and so on up a chain. Its most distant ancestor is a full-fledged math type called its *base type.*

The first use of math subtypes is to serve as mathematical models for program types whose model values are constrained to lie in some subset of the base type's set. Partial_Map_Template [Edwards 94] gives an example of this use.

The other use of math subtypes is to make it possible to be explicit about intent when defining a math operation that otherwise might be thought of as only partially defined. If a math operation is constrained by a theory or by its definition precisely where certain math subtype constraints hold, then you may declare its formal parameters to be of the corresponding math subtypes. Without math subtypes, you would have to say that the parameters are of the base types of these math subtypes. We emphasize that there is *no difference* between these two cases in terms of the theory itself or in terms of possible models for it, because the meaning of RESOLVE mathematics is determined only after replacing each math subtype everywhere by its base type. However, by using math subtypes at mathematical "compile time", it becomes possible to detect and warn you about any use of a math operation in a situation where it is not possible to show that the arguments satisfy the math subtype constraints. In a sense, then, math subtypes with their constraints serve the same purpose for a math operation as a precondition does for a program operation: Both describe when it is sensible to use an operation.

In the example above, for instance, you might define a math subtype "non-zero-rational" with the constraint "$r \neq 0$". Then the formal parameter of "reciprocal" can be declared as a non-zero-rational, not a rational. This information is sufficient to determine that "reciprocal(0)" is a potentially misleading expression. Again, there is nothing wrong with this expression mathematically, but from the standpoint of human understanding it leaves something to be desired.

Section 3 explains how you declare various new theory-related entities, including math subtypes. Before we discuss this, however, we introduce the built-in mathematical notation used throughout RESOLVE [Edwards 94, Bucci 94].

2. Built-in Mathematical Notation

RESOLVE includes an assertion sublanguage which we define here informally by listing prototypes of the built-in RESOLVE constructs, along with their counterparts from Gries and Schneider's *A Logical Approach to Discrete Math* (hereinafter, GS notation).[4]

---

3 Dahl [Dahl 92] calls the same notion a "semantic subtype".

4 Gries and Schneider [Gries 93] give axioms for various theories, precedence rules, etc. We don't necessarily use precisely the same axioms, but we do adopt the same rules for precedence and other convenience features such as "conjunctional" operators. For example, in RESOLVE as in GS notation, you may write `x < y < z` as shorthand for `x < y` **`and`** `y < z`.

RESOLVE notation is considerably less compact than GS notation. There are two main reasons we prefer it nonetheless:

- More compact notations tend to use symbols, such as ∃, that are not standard ASCII characters. This makes it difficult to build portable tools that process mathematical notation.

- You only get to make a first impression once. Our experience with people who are just learning to use formal methods (especially freshmen university students and non-traditional students, e.g., practicing programmers) is that it is important to be able to "pronounce" notation easily.

If and when these reasons no longer hold — say, GS or some other formal notation becomes the accepted standard for writing mathematics — we will cheerfully replace the RESOLVE assertion language with the standard notation. Right now, though, there is no standard notation, so we use GS notation here mainly as a convenient vehicle for explaining RESOLVE notation without providing further details. We emphasize that no part of RESOLVE is tied to the approach used to define any particular GS notation.

In the tables that follow, we don't explicitly state the math subtypes of the formal parameters to math operations. For example, `i1` **`mod`** `i2` is not constrained by integer theory when $i2 \leq 0$. In a model of integer theory, the **`mod`** operator maps to a total function — it's just that you don't care or know anything about that function's value when $i2 \leq 0$. Full details of the built-in notation involve using math subtypes to express such facts, but we trust that common knowledge and the connection to GS notation permit the current approach to suffice for this presentation.

2.1. Propositional Logic

Propositional logic notation is the backbone of the assertion notation, since assertions are **`boolean`**-valued expressions. The only unusual RESOLVE notations in this area are **`if-then`** and **`if-then-else`** — which some other specification languages also use and which help to organize and to clarify some assertions. Note that an **`if-then-else`** expression is **`boolean`**-valued, so it can't be used like this:

```
i = (if k < 0 then 1 else -1) * j
```

RESOLVE Notation	GS Notation
boolean	bool
true, false	true, false
not x	$\neg x$
x **and** y	$x \wedge y$
x **or** y	$x \vee y$
x **xor** y	$\neg(x \equiv y)$
x **iff** y	$x \equiv y$
if x **then** y or x **implies** y	$x \Rightarrow y$
if x **then** y **else** z	$(x \Rightarrow y) \wedge (\neg x \Rightarrow z)$

2.2. Predicate Calculus (With Equality)

All theories used in RESOLVE include equality, with the usual properties of reflexivity, symmetry, transitivity, and substitutivity. Note that "=" and "/=" are overloaded, i.e., they are different math operations when used with different math types. Some other RESOLVE notations are overloaded, too; math type information can disambiguate.

Each quantification schema in the table below shows a list of dummy (quantified) variables containing a single variable and its math type. This list actually may contain more than one variable of a given math type, and variables of more than one math type. For example, you may write:

```
for all x, y: MT1, z: MT2 (P)
```

Here, x and y have math type MT1 and z has math type MT2. In GS notation, you don't need to state the type of each variable introduced in a quantification; in RESOLVE you *must* provide each variable's math type.

A useful feature of the RESOLVE notation is the ability to define math subtypes, as discussed in Section 1.3. Every place a dummy variable may be introduced with its math type, you may use a math subtype instead. Suppose ST is a math subtype of math type MT with constraint C (expressed in terms of exemplar x; see Section 3.3.1). Then:

```
for all x: ST (P)
```

is equivalent to:

```
for all x: MT where (C) (P)
```

and you may use either form.

RESOLVE Notation	GS Notation
`x = y`	x = y
`x /= y`	¬(x = y)
`for all` `x: MT (P)`	(∀ x: MT | : P)
`for all` `x: MT` **`where`** `(R) (P)`	(∀ x: MT | R : P)
`there exists` `x: MT (P)`	(∃ x: MT | : P)
`there exists` `x: MT` **`where`** `(R) (P)`	(∃ x: MT | R : P)

2.3. Integers

Integer theory is built-in, but there are no math types called, e.g., "natural", "positive", or "negative", because these would be considered math subtypes of **`integer`**. We show only representative relational operators in the table. And note that some math operations (**`mod`**, **`div`**, and `^`) are unconstrained by integer theory for certain values of their parameters but nonetheless are total, as explained in Section 1.3.

GS notation includes an elegant uniform way of writing a "quantification" for any commutative, associative binary operator. RESOLVE has built-in versions of several instances that are most useful. For integers these are **`sum`** and **`product`**. As in GS, of course, these schema apply only when the expression E is of type **`integer`**. If the range of quantification R is **`true`**, then you may omit the **`where`** `(R)` clause. Where the table shows one dummy variable, you may use a list of dummy variables; and you may use math subtypes for dummies, as in Section 2.2.

RESOLVE Notation	GS Notation
integer	integer
`..., -2, -1, 0, 1, 2, ...`	..., –2, –1, 0, 1, 2, ...
`-i`	–i
`i1 + i2`	i1 + i2
`i1 - i2`	i1 – i2
`i1 * i2`	i1 • i2
i1 **div** i2	i1 ÷ i2
i1 **mod** i2	i1 **mod** i2
`i1 ^ (i2)`	$i1^{i2}$
`i1 < i2`	i1 < i2
`i1 <= i2`	i1 ≤ i2
sum x: MT **where** (R) (E)	(+ x: MT | R : E)
product x: MT **where** (R) (E)	(• x: MT | R : E)

2.4. Tuples

Tuples, or Cartesian products (ordered pairs, triples, etc.), provide the first example of math type *constructors*, where an entire family of math types is explainable through a schema that has math types as parameters. In the table below, the parameters are called MT1 and MT2, as we show only ordered pairs. In general, tuples can have arbitrary (fixed and finite) dimension.

In RESOLVE, the components of tuples have names, just as programming language records have field names. The usual dot notation is used for component extraction.

When declaring a math type using tuple notation, you may separate the component designators (e.g., `c1: MT1` and `c2: MT2`) from one another with just white space; or you may separate them with commas if you wish. A tuple-valued expression (e.g., `(x, y)`) must include commas to separate the component values because these might themselves be complex expressions containing white space.

RESOLVE Notation	GS Notation
`(c1: MT1 c2: MT2)` or `(c1: MT1, c2: MT2)`	MT1 $\times$ MT2
`t.c1`	no comparable notation
`(x, y)`	$\langle$ x, y $\rangle$

2.5. Functions and Relations

Functions (and relations, which are treated as **`boolean`**-valued functions) give you a very powerful modeling and specification tool. Function theory is a schema by which you can define a family of math types, each member of which results from fixing two math type parameters. The functions here are not precisely the same as the ones a model associates with math operations. The latter are functions on some universe of discourse. The functions described in this section are *values* in the universe of discourse to which a model maps the math type **`function from`** `MT1` **`to`** `MT2` for given MT1 and MT2.

Just as a model for a theory maps every math operation to a total function, in the standard model we have in mind for function theory every **`function`** is total. If you need to talk about a partial function in a specification, there are two basic approaches. The first is to define an appropriate math subtype to be the domain of a **`function`**. But this only works when the domain can be characterized in advance. The second approach — which we use, for example, in specifying the programming type Partial_Map [Edwards 94], where the domain of the partial function can change dynamically under program control — is to treat a partial function as a set of ordered pairs with the "function property". Again, a math subtype is involved: a math subtype of **`set of`** `(x: MT1 y: MT2)`.

The notation **`differ`** `(f1, f2, s)` says that f1 and f2 *might* differ on subset s of their common domain; they agree everywhere else. GS notation includes an "alter" function for the sequence type, but we find **`differ`** to be more flexible and useful for writing specifications.

RESOLVE Notation	GS Notation
`function from` `MT1` **`to`** `MT2`	MT1 $\rightarrow$ MT2
`f(x)`	f.x
`f^(k)(x)`	$f^k.x$
`differ` `(f1, f2, s)`	($\forall$ x: MT1 | x $\notin$ s : f1.x = f2.x) where f1, f2: MT1 $\rightarrow$ MT2

2.6. Sets and Multisets

RESOLVE notation for sets and multisets should be clear. Note that the notation |s| denotes the cardinality of s; and that the schemas for set comprehension (borrowed from GS notation) are powerful, but it takes a bit of practice to learn to use and to read them.

Although the table does not show these cases, you also may use **`is not`** in place of **`is`**, **`subset`** in place of **`superset`**, and/or **`proper`** before **`subset`** or **`superset`**, with the usual meanings.

RESOLVE Notation	**GS Notation**
`finite set of` `MT`	set(MT)
`empty_set`	{ }
`universal_set`	**U**
`{x, y, z}`	{x, y, z}
`{x: MT` **`where`** `(P)}`	{x: MT | P}
`{x: MT` **`where`** `(P) (E)}`	{x: MT | P : E}
`x` **`is in`** `s`	$x \in s$
`s` **`union`** `t`	$s \cup t$
`s` **`intersection`** `t`	$s \cap t$
`s - t` or `s` **`without`** `t`	s – t
`s` **`is subset of`** `t`	$s \subseteq t$
`|s|`	#s
`union` `x: MT` **`where`** `(R) (E)`	($\cup$ x: MT | R : E)
`intersection` `x: MT` **`where`** `(R) (E)`	($\cap$ x: MT | R : E)

For sets that are not necessarily finite, the type declaration is slightly different (it does not iclude the word **`finite`**) and there are two additional notations.

RESOLVE Notation	**GS Notation**
`set of` `MT`	set(MT)
`s` **`is finite`**	$\#s < \aleph_0$
`s` **`is countable`**	$\#s \leq \aleph_0$

There are also the types **finite multiset** and **multiset**, which are collections of elements that (unlike those in sets) are not necessarily distinct. GS notation includes "bag" as the term for multiset. The RESOLVE set notations apply to multisets, too, with the obvious meanings; see GS for details.

2.7. Strings

String theory (or what GS notation calls sequence theory) is very important and convenient for specifying ADTs, so it is built-in. The main notations here are explicit string construction from individual elements of type MT, e.g., <x1, x2, x3>; s1 * s2, the concatenation of s1 and s2; and |s|, the length of s. Note that RESOLVE strings are finite, as are GS sequences.

In RESOLVE, there is no special notation for picking out the i^{th} element of a **string**. If you need to say this then we recommend that you use a **function from integer to** `MT`. Or, if you wish, you can define a math operation to pick out **string** elements; see Section 3.3.4.

Although the table does not show these cases, you also may use **is not** in place of **is**, **suffix** in place of **prefix**, **superstring** in place of **substring**, and/or **proper** before **substring** or **superstring** or **prefix** or **suffix**, with the obvious meanings.

A useful additional notation, in our experience, is **elements** `(s)`, the **finite set** of elements that occur in s at least once.

RESOLVE Notation	**GS Notation**
string of `MT`	seq(MT)
empty_string	∈
`<x1, x2>`	<x1, x2>
`s1 * s2`	s1 ^ s2
`|s|`	#s
`s1` **is prefix of** `s2`	isprefix(s1,s2)
`s1` **is substring of** `s2`	isseg(s1,s2)
reverse `(s)`	rev(s)
occurs_count `(s, x)`	x # s
elements `(s)`	no comparable notation
`s1` **is permutation of** `s2`	perm(s1,s2)

3. Math Modules

A *math module* is a RESOLVE unit that defines some (reusable) mathematics. You should consider most of the built-in mathematical theories and notation of RESOLVE, discussed in Section 2, to be recorded in special math modules that are so important that they are built-in with syntactic sugar. But, of course, not all useful mathematics can be built-in. So we provide a way to write down new theories — for the relatively rare occasions when they are needed — and additions to existing ones, both for human use and for use by various tools such as a program verifier.

The structure of a math module mirrors the structure of the other RESOLVE modules. Especially notice the parallels to conceptual modules, which are not accidental: The problem of writing down a formal statement of mathematical ideas is almost like the problem of writing down a formal statement of programming ideas. So, for consistency, we use similar keywords for similar ideas in the various RESOLVE modules. Because of the many connections, you should find that having read about specifying concepts [Edwards 94] makes this section much easier to understand.

For all the parallels between conceptual and math modules, a natural question is why both are needed. The primary reason is separation of concerns, as the developers of Larch's "two-tiered" approach to formal specification have explained well. Mathematical and programming ideas are similar and the notational requirements analogous, but they are not identical. There are a variety of technical differences in RESOLVE, e.g., the treatment of type matching [Harms 90, Harms 91]. So we also adopt a two-tiered approach and keep modules that define mathematics separate from modules that specify program behavior. (In fact, we have a three-tiered approach, counting realizations.)

A RESOLVE math module can define three major kinds of identifiers, *math type families*, *math subtype families*, and *math operation families*. All RESOLVE math modules are generic, in the same sense as for program modules [Edwards 94, Bucci 94]. This means that a client must instantiate a math module before using it. An instance of a math module is called a *math facility*. A typical math module — but not every one — defines a math type family, possibly some math subtype families, and some math operation families. Each instance of it defines a particular *math type*, some particular *math subtypes*, and some particular *math operations* which you obtain, in effect, by replacing the formal generic parameters with the actuals used in creating that instance. As with their programming counterparts, when the difference between a particular math type/math subtype/operation and the family of which it is a member is clear from context, we omit the word "family".

Figure 1 shows the terminological parallels between math modules and conceptual modules, in order to set up the general picture. Notice that there is no programming counterpart for math subtypes.

Figure 1: Math Module and Conceptual Module Terminology

Math Module	**Conceptual Module**
math type family	type family
math type	type
math operation family	operation family
math operation	operation
math facility	facility

We recommend that you organize math modules into the three main categories suggested for conceptual modules: kernels, additions, and enhancements [Edwards 94]. Figure 2 shows a kernel math module, i.e., one that defines a new theory. In this case the module defines a theory of binary trees, which is useful but not built-in to RESOLVE. Figures 3 and 4 define additions to this theory, i.e., they define additional math operations and/or state theorems that might be handy for proving assertions involving the new math operations. We do not show an example enhancement, but you can combine math modules in the same way as you can combine conceptual modules [Edwards 94]. This is most useful if there are several additions to a theory that might be reused together in many places.

Figure 2: A New Theory (A Kernel Math Module)

```
mathematics BINARY_TREE_THEORY_TEMPLATE

   context

      parametric context

         math type LABEL

   interface

      math type TREE

      math operation EMPTY_TREE: TREE

      math operation COMPOSE (
            root: LABEL
            left: TREE
            right: TREE
         ): TREE

      axiom 1 is
         for all root: LABEL, left, right: TREE
            (COMPOSE (root, left, right) /= EMPTY_TREE)

      axiom 2 is
         for all root1, root2: LABEL,
               left1, left2, right1, right2: TREE
            (if     COMPOSE (root1, left1, right1) =
                    COMPOSE (root2, left2, right2)
             then root1 = root2  and
                    left1 = left2  and
                    right1 = right2)

      axiom induction is
         for all s: set of TREE, root: LABEL, left, right: TREE
            (if     EMPTY_TREE is in s  and
                    if left is in s  and
                         right is in s
                    then COMPOSE (root, left, right) is in s
             then s = universal_set)

end BINARY_TREE_THEORY_TEMPLATE
```

Figure 3: A New Math Operation and Theorem (An Addition)

```
mathematics BINARY_TREE_SIZE_MACHINERY

   context

      global context

         mathematics BINARY_TREE_THEORY_TEMPLATE

      parametric context

         math type LABEL

         math facility BINARY_TREE_THEORY is
            BINARY_TREE_THEORY_TEMPLATE (LABEL)

   interface

      math operation SIZE (
            t: TREE
      ): integer
         implicit definition
            for all root: LABEL, left, right: TREE
               ((if    t = EMPTY_TREE
                 then  SIZE (t) = 0)  and
                (if    t = COMPOSE (root, left, right)
                 then  SIZE (t) = SIZE (left) +
                          SIZE (right) + 1))

      theorem non_negative_size is
         for all t: TREE (SIZE (t) >= 0)

end BINARY_TREE_SIZE_MACHINERY
```

Figure 4: Another Addition

```
mathematics BINARY_TREE_SUBTREE_MACHINERY

  context

    global context

      mathematics BINARY_TREE_THEORY_TEMPLATE

    parametric context

      math type LABEL

      math facility BINARY_TREE_THEORY is
        BINARY_TREE_THEORY_TEMPLATE (LABEL)

  interface

    math subtype NON_EMPTY_TREE is TREE
      exemplar    n
      constraint n /= EMPTY_TREE

    math operation LEFT_SUBTREE (
        t: NON_EMPTY_TREE
      ): TREE
      implicit definition
        there exists root: LABEL, right: TREE,
          (t = COMPOSE (root, LEFT_SUBTREE (t), right))

    math operation RIGHT_SUBTREE (
        t: NON_EMPTY_TREE
      ): TREE
      implicit definition
        there exists root: LABEL, left: TREE,
          (t = COMPOSE (root, left, RIGHT_SUBTREE (t)))

end BINARY_TREE_SUBTREE_MACHINERY
```

3.1. High-Level Structure of a Math Module

A math module, like other RESOLVE modules, has a context section and an interface section:

```
<math_module> ::=
  mathematics <math_module_id>
     [<math_context_section>]
     <math_interface_section>
  end <math_module_id>
```

3.2. Context Section

A math module can have two different kinds of context, global and parametric:

```
<math_context_section> ::=
  context
     [<math_global_context_section>]
     [<math_parametric_context_section>]
```

Visibility, scope, and name qualification rules are the same as for conceptual modules [Edwards 94]. Namely, when you introduce an instance of a math module into a context, you make visible all identifiers that are declared in its interface *and* those visible through or declared in its context. Overloading of identifiers is permitted, with no qualification required unless an ambiguity arises.

Notice that there is no local context section in a math module. Alternatively, we could have left out the interface section. The reason is that there can be no difference between declarations made in the local context section and the interface section, because there is no need here to distinguish between mathematical and programming identifiers as there is for conceptual modules [Edwards 94]. *All* identifiers in math modules are mathematical identifiers and are implicitly exported by virtue of being in the context.

3.2.1. Global Context

Global context introduces dependence on other math modules or math facilities:

```
<math_global_context_section> ::=
  global context
     <math_global_context_item_sequence>

<math_global_context_item> ::=
  mathematics <math_module_id> |
  math facility <math_facility_id>
```

The only reason for coupling a math module to another math module is that you need to describe some instance of the latter in the parametric context of the former, as discussed below. As a practical matter, any new theory that you might add to RESOLVE (e.g., as in Figure 2) is likely to be independent of other particular theories and their instances, except perhaps built-in ones. And since you don't mention built-in theories explicitly, global context should rarely appear in a math module that defines a new theory.

On the other hand, you might want to enhance a theory by making explicit definitions for additional math operations that don't participate in the axioms of the theory. (You may not add axioms in an addition or enhancement.) You also might want to list some theorems to which a client of a theory might appeal in proofs involving that theory. Here, you use global context and parametric context to connect new math operations and theorems to underlying theories. You do this in a stylized way that parallels the method for making additions to a kernel concept [Edwards 94].

3.2.2. Parametric Context

In parametric context you list the generic formal parameters of a math module along with restrictions on them, if any:

```
<math_parametric_context_section> ::=
   parametric context
      <math_parameter_sequence>
      [<math_parametric_context_restriction>]
```

There are three kinds of parameters: math types, math operations, and math facilities:

```
<math_parameter> ::=
   <math_type_parameter> |
   <math_operation_parameter> |
   <math_facility_parameter>

<math_type_parameter> ::=
   math type <math_type_id>

<math_operation_parameter> ::=
   math operation <math_operation_id>
      [(<math_operation_formal_parameter_sequence>)]:
         <math_type_id>

<math_facility_parameter> ::=
   math facility <math_facility_id> is <math_module_id>
      [(<math_facility_argument_list>)]
```

A math type formal parameter simply introduces a local name for the (unknown) math type that a client must provide to instantiate the math module. For a math type parameter, the client may provide any math type as the corresponding actual.

A math operation formal parameter introduces a local name in the same way, but also constrains the actual. For a math operation parameter, the client must provide a math operation with the same math type signature as the formal — after substitution of all math type actuals for the corresponding formals.

For a math facility parameter, a client must provide some instance of the named math module whose parameters match those in the formal's declaration.

There is an optional **`restriction`** clause at the end of the parametric context section:

```
<math_parametric_context_restriction> ::=
  restriction <assertion>
```

This allows you to constrain the actual parameters used to instantiate a math module so that they must satisfy some properties. For example, you might want to insist that the actual passed for a formal which is a math operation — and which has the declared structure of a binary relation — has the properties of an equivalence relation.

3.3. Interface Section

The interface section is the second major part of a math module:

```
<math_interface_section> ::=
   interface
      [re-exports
         <math_re-exported_item_sequence>]
      [<math_interface_item_sequence>]

<math_re-exported_item> ::=
   <math_facility_id>
      [<math_renaming_section>]

<maht_renaming_section> ::=
   renaming
      <math_renaming_item_sequence>

<math_renaming_item> ::=
   <math_type_id> as <math_type_id> |
   <math_operation_id> as <math_operation_id> |
   <math_axiom_id> as <math_axiom_id> |
   <math_theorem_id> as <math_theorem_id> |

<math_interface_item> ::=
   <math_type_declaration> |
   <math_subtype_declaration> |
   <math_operation_header_declaration> |
   <math_axiom_declaration> |
   <math_operation_definition_declaration> |
   <math_theorem_declaration>
```

The interface section for a new theory declares math types (usually one), math subtypes (usually none), and headers for math operations that appear in the axioms. These math operations are comparable to primary operations exported by a conceptual module in that they are “essential”.

A math module also can explicitly define additional math operations and can state theorems that follow from the axioms. These math operations are comparable to secondary operations on the programming side. They add nothing to a theory; in particular, they do not affect which interpretations of the theory are models. But they can make it more convenient for you to express assertions. To highlight the difference between axiomatically-defined and explicitly-defined math operations and the difference between axioms and theorems, we highly recommend that you separate explicitly-defined math operations (and theorems involving them) into separate math modules that are additions to kernel math modules, as illustrated in Figures 3 and 4. You also may combine a kernel math module and one or more additions into an enhancement, using the **`re-exports`** feature and optional **`renaming`**, in precisely the same way as for conceptual modules [Edwards 94].

3.3.1. Math Types and Subtypes

When declaring a math type (family), you simply give it a name:

```
<math_type_declaration> ::=
   math type <math_type_id>
```

To define a math subtype, you may provide a name which acts as a synonym for another math type or subtype, or you may constrain the values taken by variables of the new subtype:

```
<math_subtype_declaration> ::=
   math subtype <math_type_id> is <math_type_id>
      [<math_subtype_exemplar>
       <math_subtype_constraint>]

<math_subtype_exemplar> ::=
   exemplar <math_variable_id>

<math_subtype_constraint> ::=
   constraint <assertion>
```

The **`exemplar`** declaration introduces an identifier to use in the **`constraint`** clause to stand for a value of the math subtype. As discussed in Section 1.3, even when you require a math subtype's values to satisfy a **`constraint`**, you may still use the math subtype anywhere you could use its base type, to give the reader a clear hint about your intentions. By doing this you impose a "warning level" proof obligation to show all occurrences of variables of that math subtype occur in situations where the constraint is satisfied; i.e., the hint is not misleading.

3.3.2. Essential Math Operations

When declaring a math operation that participates in the axioms of a theory, you give only its header, just as you do for a math operation parameter:

```
<math_operation_header_declaration> ::=
   math operation <math_operation_id>
      [(<math_operation_formal_parameter_sequence>)]:
         <math_type_id>

<math_operation_formal_parameter> ::=
   <tuple_component_id> : <math_type_id>
```

The math type signature you establish in the header of a math operation allows you to screen out nonsense. This way we don't have to worry about giving meanings to assertions that are not well-formed from the usual syntactic or math typing standpoints. Furthermore, you can distinguish among many possibilities for overloaded names. These uses of math types are entirely analogous to the use of programming types in a strongly-typed programming language.

3.3.3. Axioms

Each axiom has an identifier as well as some content:

```
<math_axiom_declaration> ::=
  axiom <axiom_id> is <assertion>
```

The assertion in an axiom contains dummies whose scope is local to that assertion. When writing axioms, it is standard practice to have free variables, which are implicitly universally quantified. RESOLVE requires that you explicitly quantify each dummy and give its math type. This provides some protection against easily detectable typographical errors, among other things.

3.3.4. Additional Math Operations

You declare an additional math operation, i.e., one defined explicitly and not via the axioms, by giving both its header and an assertion that defines it:

```
<math_operation_definition_declaration> ::=
  <math_operation_header_declaration>
    [explicit | implicit] definition <assertion>
```

In the assertion, typically you write a direct equational (“explicit”) definition. But sometimes you need to define a math operation indirectly (“implicit”), saying that the value of the math operation satisfies some property that is not expressed equationally. Examples of this include LEFT_SUBTREE and RIGHT_SUBTREE in Figure 4. A special case is an inductive definition, e.g., SIZE in Figure 3. In any event, any new math operation you declare raises a proof obligation: it must be well-defined. Part of this demonstration involves showing that math subtype constraints are satisfied, as discussed in Section 1.3.

3.3.5. Theorems

Each theorem, like each axiom, has an identifier as well as some content:

```
<math_theorem_declaration> ::=
  theorem <theorem_id> is <assertion>
```

Of course, a proof obligation is raised if you state a theorem: the theorem must indeed be provable from the axioms.

You can use a theorem to document a fact that explains how an algorithm works, enhancing both human understanding and the ability of a tool to analyze your program automatically. Suppose you write a recursive (program) operation to reverse a FIFO Queue, which is modeled as a string. Then the following theorem explains *why* the code works, and plays a central role in its correctness proof:

```
theorem reverse_of_concatenation is
   for all s1, s2: string of ENTRY
      (REVERSE (s1 * s2) = REVERSE (s2) * REVERSE (s1))
```

This theorem is a fact about strings that has nothing to do with FIFO Queues. It can be reused elsewhere, and should be recorded outside any particular realization where it happens to be used. A RESOLVE math module (parameterized by ENTRY) is where you'd put it.

4. Conclusion

The mathematical foundations of RESOLVE have played a key role as we have settled on most of our language features and component engineering principles. By trying to make sure there is a firm formal basis for modular verification in the face of each potential new language construct or design principle, we have discovered many subtle problems with related work and have avoided many pitfalls. Formal methods can pay off as part of such a research approach even in the unfortunate event that no one seriously considers using them as part of everyday software engineering activities.

5. Acknowledgment

Many people have contributed to this work, but we especially thank Steve Edwards, Doug Harms, Joe Hollingsworth, Joan Krone, Murali Sitaraman, and Stu Zweben.

We also gratefully acknowledge the financial support for our research from the National Science Foundation, under grant CCR-9311702; and from the Advanced Research Projects Agency of the Department of Defense, under ARPA contract number F30602-93-C-0243, monitored by the USAF Materiel Command, Rome Laboratories, ARPA order number A714.

6. References

[Bucci 94] Bucci, P., Hollingsworth, J.E., Krone, J., and Weide, B.W., "Implementing Components in RESOLVE", *Softw. Eng. Notes 19*, 4 (Oct. 1994), to appear.

[Dahl 92] Dahl, O-J., *Verifiable Programming*, Prentice-Hall International (UK) Ltd., 1992.

[Edwards 94] Edwards, S.H., Long, T.J., Sitaraman, M., and Weide, B.W., "Specifying Components in RESOLVE", *Softw. Eng. Notes 19*, 4 (Oct. 1994), to appear.

[Gries 93] Gries, D., and Schneider, F.B., *A Logical Approach to Disrcete Math*, Springer-Verlag, 1993.

[Guttag 93] Guttag, J.V., and Horning, J.J., *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, 1993.

[Harms 90] Harms, D.E., *The Influence of Software Reuse on Programming Language Design*. Ph.D. diss., Dept. of Comp. and Inf. Sci., Ohio State Univ., Columbus, Aug. 1990. Available from UMI (phone: 800-521-0600).

[Harms 91] Harms, D.E., and Weide, B.W., "Copying and Swapping: Influences on the Design of Reusable Software Components", *IEEE Trans. on Software Eng. 17*, 5 (May 1991), 424-435.

[Ogden 94] Ogden, W.F., Sitaraman, M., Weide, B.W., and Zweben, S.H., "The RESOLVE Framework and Discipline — A Research Synopsis", *Softw. Eng. Notes 19*, 4 (Oct. 1994), to appear.

[Spivey 89] Spivey, J.M., *The Z Notation*, Prentice-Hall International (UK) Ltd., 1989.