Unique Names Violations, a Problem for Model Integration
or
You Say Tomato, I Say Tomahto[*]

Hemant K. Bhargava
AS/BH
Naval Postgraduate School
Monterey, CA 93943-5000

Steven O. Kimbrough
University of Pennsylvania
Department of Decision Sciences/6366
Philadelphia, PA 19104

Ramayya Krishnan
SUPA
Carnegie Mellon University
Pittsburgh, PA 15213

October 13, 1990

[*] Apologies to George and Ira Gershwin. Thanks to the U.S. Coast Guard and contract DTCG39–86–C–80348, between the U.S. Coast Guard and the University of Pennsylvania with Steven O. Kimbrough as principal investigator, which funded this work in part. Thanks, too, to Christopher V. Jones for comments on an earlier draft and for contributing so essentially to an atmosphere in which model management is discussed with excitement.

Abstract

The tomato-tomahto problem (known as the synonymy problem in the database literature) arises in the context of model management when different names are used in different models for what should be identical variables, and these different models are to be integrated or combined into a larger model. When this problem occurs, it is said that the unique names assumption has been violated. We propose a method by which violations of the unique names assumption can be automatically detected. The method relies on declaring four kinds of information about modeling variables: dimensional information, laws relating dimensional expressions, information (called the *quiddity*) about the intended interpretation of the variables, and laws relating quiddity expressions. We present and discuss the method and the principles and theory behind it, and we describe our (prototype) implementation of the method, as an additional function of an existing model management system.

file: tomato-tomahto

1 Introduction

Batman is Bruce Wayne. Clark Kent is Superman. Cicero is Tully. Plato is Aristocles. Phosphorus, the morning star, is Hesperus, the evening star. While it is unusual, it is certainly possible for one thing to have more than one name. This occurs in fiction ('Batman' and 'Bruce Wayne' name the same individual), in ordinary discourse ('Cicero' and 'Tully' are two names for the same historical person), and in science (both 'Phosphorus' and 'Hesperus' name the planet Venus). That every individual has at most one name, unless stated otherwise, is often a useful and convenient assumption in software systems, and is called *the unique names assumption* [16].[1]

Unique names violations may occur for several reasons, including: intention to deceive (e.g., Batman, Superman); whimsy (e.g., nick names, 'Plato' roughly meaning chubby and being a nick name given to Aristocles by his wrestling coach); and error or inadvertence (e.g., Hesperus and Phosphorus). Our concern in this paper is with the consequences for model management, particularly model integration, of unique names violations. We shall now illustrate with a very simple example the problem such violations—usually due to error or inadvertence—can create.

Consider that we are building a model of the cost of a shipment composed of ketchup and cocktail sauce. The resulting integrated model is to be:

$$c_{Total} = c_k \cdot k + c_c \cdot c \tag{1}$$

where c_{Total} is the total cost of the shipment, c_k the unit cost of ketchup, k the number of units of ketchup, c_c the unit cost of cocktail sauce, and c the

[1] Unique names is implied by, and is a special case of *the closed-world assumption.* Either assumption can be used for nonmontonic reasoning systems.

number of units of cocktail sauce. You build a model of the cost of the ketchup c_k, I build a model of the cost of cocktail sauce, c_c. Because both ketchup and cocktail sauce are made out of tomatoes, both models must take account of the cost of tomatoes. But, we happen to use different variable names. You say *Tomato,* I say *Tomahto.* Both variables refer to the same thing—the unit cost of tomatoes—and any assumption of unique names is violated.

Do we have a problem? Well, in the integrated model, we have two names for the same thing. Certainly, we require that in any instantiation of the model the two variables should have the same value, i.e.,

$$\mathit{Tomato} = \mathit{Tomahto} \tag{2}$$

Notice, however, that nothing in the mathematics of our model—1—requires that 2 be satisfied. Further, we note that it would be quite easy in an implementation for *Tomato* and *Tomahto* to have different values. You did your estimate, I did mine. Suppose the two were different. Then the resulting joint model is clearly invalid. Even if the two estimates are not presently different, data values may change over time, so it is fair to say that the joint model is at risk. Good practice would indicate a correction.

What are we to do? The song says, "Let's call the whole thing off." Less draconian action will remedy the problem. We simply need to identify the non-unique names, agree to replace them with a unique name, and agree to what the value of that variable should be. Doing this will resolve the problem. The remaining questions are concerned with how exactly to do this with maximal machine-based support. What can a model management system provide by way of automated support for resolving unique names violations? The main purpose of this paper is to investigate principles, and to develop techniques, for answering this question.

We note that the tomato-tomahto problem has arisen in the database literature. Briefly, integrated database design usually consists of view modeling, in which user requirements are formally expressed by means of one or more user-oriented schemas, followed by an integration process that merges the schemas into a global schema [1, 2]. Because different people will often refer to the same concept or data without knowing how others will do the same, database integration has to deal with naming conflicts. Two types of naming conflicts have been distintuished in the literature: homonyms (the same name is used to refer to two distinct concepts) and synonyms (two distinct names are used to refer to the same concept). The synonym problem is essentially the problem of unique names violations.

Although the problem is recognized, most methodologies for database integration assume that the synonym problem (unique names violations) is dealt with prior to integration [11, 18]. Others (e.g., [12]) suggest that "Naming conflicts are easily handled by renaming," without proposing how such conflicts are to be discovered. Batini and Lenserini [1] and ElMasri et al. [14] propose a strategy for discovering unique names violations by assigning a “degree of similarity” to concepts with the same names. The strategy proposes the heuristic use of information about types, constraints (e.g., cardinality of entity sets), and membership in relationships to identity problems. These pieces of information are referred to as “indications” and require the designer to analyze each indication to discover unique names violations. We have not been able to find literature on computer-based support for identifying problems of unique names violations.

The remainder of the paper is organized as follows. In §2 we present and discuss several examples of the tomato–tomahto problem (of unique names violations) in model management. Our purpose in that section is to introduce and motivate both some of the nuances of the general problem and our solution to that problem. We give a more formal and complete presentation of our solution in §3. It is not our claim that this is a complete solution to the problem. Rather, we shall argue that our proposal is quite powerful and can be extended in fairly straightforward ways to yield yet more powerful means of dealing the the problem of unique names violations. In §4, we present an overview of our model management system, TEFA, which is in use by the U.S. Coast Guard. Following that, we decribe and discuss our implementation in TEFA of our solution to the tomato-tomahto problem. In §5, we extend the analysis and discuss the use of quiddities for semantic validation of models. We conclude, in §6, with comments about our proposed solution to the tomato-tomahto problem.

2 Example Problems for Solution

Our aim in this section is to discuss, in a preliminary fashion, examples that illustrate our solution strategy for the tomato-tomahto problem. We will also discuss certain characteristics or criteria that may be used to evaluate different solutions under this strategy. We shall present our theory much more carefully in §3. Our hope here is that by presenting and discussing some example problems and initial solutions we can motivate and clarify the theory and discussion that follows.

As noted in §1, the principal challenge for machine-based assistance on the tomato-tomahto problem is automatically to identify variables that are intended to represent the same real-world object, but that have different names in submodels (or even in different parts of a common model). Since the names are different, other kinds of information **about** the variables are required to make this identification. The essence of our strategy is to develop a principled means of supplying, and expressing, such information. Hence we are now concerned with three issues: 1) What kinds of information is relevant to the problem?, 2) How should this information be represented? and 3) What kinds of inferences can be performed to make the identification? We will now present a few informal examples designed to illustrate a preliminary solution strategy. The objective of our solutions is to identify pairs of variables that present *possible* unique names violations.

Consider the following example, in which the cost of purchasing a truck is used in two models with two different variables.

Example 1

- Model 1
  - Variable: *purchase-cost*
  - Description: "Purchase cost of a truck"
- Model 2
  - Variable: *cost-of-purchase*
  - Description: "Cost of purchase of a truck"

Given this unique names violation, what might be done by way of detecting it automatically? As the example stands, very little. Of course, it would be possible to write a program that would recognize that both *purchase-cost* and *cost-of-purchase* contain *purchase* and *cost* substrings, and hence may be intended to refer to the same thing. Similarly, a parsing program could be written to recognize that "Purchase cost of a truck" and "Cost of purchase of a truck" are sufficiently similar to indicate a possible problem. The computational cost of either of these programs would likely be prohibitive for any large-scale model management system. More fundamentally, any parsing program would have to work from a formal theory of how information about modeling variables is to be expressed. Proposing such a theory is the primary aim of this paper.

To begin, then, to describe our theory, we note that both variables have the same *dimension,* currency. Also both variables are about the same sort of real-world object, a truck. We say that both are about the same *stuff*. So our first solution is to represent explicitly each variable's dimension and stuff.

Example 1A

- Model 1
  - Variable: *purchase-cost*
  - Description: "Purchase cost of a truck"
  - Dimension: currency
  - Stuff: truck
- Model 2
  - Variable: *cost-of-purchase*
  - Description: "Cost of purchase of a truck"

- Dimension: currency
- Stuff: truck

Given this, our first heuristic for indicating possible unique names violations is

Rule 1. *If two syntactically distinct variables have the same dimension and stuff, this indicates a possible unique names violation.*

There are four significant points to be made about rule 1. First, given appropriate declarations as in example 1A, it is a simple matter to program a test for rule 1 and the computational complexity is $\mathcal{O}(n^2)$, where n is the number of variables to be tested.[2] Second, rule 1 should *not* be weakened to:

Rule 1A. *If two syntactically distinct variables have the same dimension or stuff, this indicates a (possible) unique names violation.*

Clearly, it is quite legitimate to have two variables that have the same dimension but that are about different stuff. If x is the cost of trucks, and y is the cost of tomatoes, no problem should be indicated. Similarly, no problem should be indicated by the fact that two different variables refer to the same stuff, but have different dimensions. It is, e.g., quite unexceptionable if x is the cost of a truck and y is the length of a truck.

This naturally raises the question whether reliance on rule 1 may produce a type 1 error, the error of indicating a problem when there is none. If rule 1 fires, does that guarantee a unique names violation? Our third point about rule 1 is that it is not immune from type 1 errors, although its firing does indicate a problem of some sort. Information about the variables (here the dimension and stuff values) could have been coded incorrectly. More interestingly, our descriptive apparatus (here the dimension and stuff attributes and the terms used to express them) may be insufficiently rich. For example, *purchase-cost* may be intended to represent the purchase cost of a truck of type A, while *cost-of-purchase* was intended for the purchase cost of a truck of type B. No unique names violation has occurred, but the firing of rule 1 would indicate that our descriptive apparatus is inadequate.

The possibility of type 1 errors—indication of a problem when there is none—raises the question whether reliance on rule 1 may produce a type 2 error, the error of failing to indicate a problem when there actually is one. Our fourth point about rule 1 is that it risks type 2 errors as well. The cause here is, again, inadequacy of the descriptive apparatus. Suppose, to modify the present example, we have a number of variables pertaining to truck cost. Some describe the cost of purchase, others capture the cost of maintenance, others the cost of operation, and so forth. There is a sense in which these variables are about

[2] If the variables are sorted by quiddity and dimension (see below), we can expect much faster performance in practice. Thanks to Chris Jones for suggesting that the variables be sorted in this way.

different stuff, so we have *truck-purchase, truck-maintenance,* and so on as terms describing stuff. But this opens the door to the tomato-tomahto problem. You say x's stuff is *truck-purchase,* I say y's stuff is *purchase-price-of-truck.* A unique names violation has occurred, but—under rule 1 and the present method of variable description—no violation gets indicated.

In sum, our response to example 1—explicitly declaring descriptive information about the modeling variables and using this information to trigger indicating rules—has yielded some benefits. We would, however, like to reduce the risks of type 1 and type 2 errors. Further, we have seen that a main cause of these errors is inadequacy of the descriptive apparatus. In discussing how we are to go about reducing these errors, we begin by taking up a new example.

Consider example 2, in which we have two variables that measure the cost of *purchase* and *production* of a truck.

Example 2

- Model 1
  - Variable: *purchase-cost*
  - Description: "Cost of purchasing a truck"
  - Dimension: currency
  - Stuff: truck
- Model 2
  - Variable: *production-cost*
  - Description: "Cost of producing a truck"
  - Dimension: currency
  - Stuff: truck

Rule 1 will include $\langle$*purchase-cost, production-cost*$\rangle$ in the candidate set of unique names violators, since the two variables have the same dimension and stuff. Doing so, however, is a type 1 error. The remedy is to improve our descriptive apparatus. Stuff is too crude a notion. We need to represent more than what the variable is about, what *stuff* it is about. We need to represent both what stuff a variable is about *and* what it is about the stuff that the variable is about. In the present example, both variables are about the same stuff, trucks, but one is about the purchase cost of a truck, while the other is about the production cost of a truck. These two very different things should be captured in the description of the variables.

We want, then, to capture descriptively much more information about what the variable is about. We would aim to capture its very essence, or *quiddity.*[3]

[3] From the *Oxford English Dictionary,* quiddity is "The real nature or essence of a thing; that which makes a thing what it is." Of course, our language for expressing quiddities is only a model, or approximation, of genuine quiddity, if it exists.

How might this be done? Consider the present example. Both variables are about the same stuff: trucks. They differ in what it is they represent *about* trucks. What is it about trucks they describe? Purchasing in one case and production in the other. What is it about purchasing and production that they represent? Cost, in both cases. And what about cost? Nothing else. This line of reasoning suggests the following means of description for the current example.

Example 2A

- Model 1
  - Variable: *purchase-cost*
  - Description: "Cost of purchasing a truck"
  - Dimension: currency
  - Quiddity: *cost(purchase(truck))*
  - Quiddity Paraphrase: "the cost of purchase of a truck"
- Model 2
  - Variable: *production-cost*
  - Description: "Cost of producing a truck"
  - Dimension: currency
  - Quiddity: *cost(production(truck))*
  - Quiddity Paraphrase: "the cost of production of a truck"

Given this, our second heuristic for indicating possible unique names violations is

Rule 2. *If two syntactically distinct variables have the same dimension and quiddity, this indicates a possible unique names violation.*

Rule 2 will not commit a type 1 error with respect to example 2A. There is no unique names violation and none is indicated. The four points made with respect to rule 1 are pertinent here as well. First, assuming a fast means of testing quiddities for identity (e.g., string matching), the required program to produce candidate violators is again easy and of complexity at most $\mathcal{O}(n^2)$. Second, the *and* in the rule should not be replaced with an *or*. Third and fourth, the possibility remains of errors of types 1 and 2.

Two additional comments are in order. Fifth, quiddity is now expressed as a logical term, either as a constant (as stuff) or as a function (as in example 2A), and is a referring expression. The move to a functional expression provides excellent flexibility and expressive power. For example, we can quite naturally express additional information about the intended meaning of the variables:

Example 2B

- Model 1
  - Variable: *labor-production-cost*
  - Description: "Cost of labor in producing a truck"
  - Dimension: currency
  - Quiddity: *cost(labor(production(truck)))*
  - Quiddity Paraphrase: "the cost of labor in production of a truck"
- Model 2
  - Variable: *materials-production-cost*
  - Description: "Cost of materials in producing a truck"
  - Dimension: currency
  - Quiddity: *cost(materials(production(truck)))*
  - Quiddity Paraphrase: "the cost of materials in production of a truck"

A primary aim of what follows is to explore how to take advantage of the flexibility and expressive power available through functional representation of quiddities and dimensions.

Our sixth comment about rule 2 and the revised, functional means of expressing quiddity, is to note that a certain ambiguity remains. In what order should functions be applied? Should, for example, the quiddity of a variable representing the cost of a truck be represented as *cost(truck)* or as *truck(cost)*? Should we have *cost(materials(production(truck)))* or *materials(cost(production(truck)))*? If you do it one way and I do it the other, we are open to type 2 errors. There are two sorts of things that can done. Both are discussed further in §3. First, we might find a way of stipulating the validity conditions of quiddity expressions so as to eliminate, or reduce, the possibility of ambiguity. Second, we might introduce equivalence transforms and a modification of rule 2 to:

Rule 3. *If two syntactically distinct variables have the same or equivalent dimension and quiddity, this indicates a possible unique names violation.*

To illustrate, we can express the fact that *cost(truck)* and *truck(cost)* are equivalent with:

$$\mathit{cost(truck)} = \mathit{truck(cost)} \tag{3}$$

(Equation 3 is admittedly *ad hoc.* We discuss generalizations in §3.)

The problem with this second strategy is that it can substantially increase the computational complexity of searching for candidate unique names violators. We note, however, that while an integrated model may be executed very many times, checking for unique names violations is only done once per integrated model.

We conclude this section with one more example for motivating our (partial) solution. The variables we have treated so far have not been subscripted. Consider example 3.

Example 3

- Model 1
  - Variable: c_{ij}
    - Description: "Cost of purchasing product j at market i"
    - Dimension: currency
    - Quiddity: *cost(purchase(arg(c(i,j)-i),arg(c(i,j)-j)))*
    - Quiddity Paraphrase: "the cost of purchasing a given product at a given market"
  - Variable: *c(i,j)-i*
    - Description: "a market"
    - Dimension: 1 (i.e., no dimension)
    - Quiddity: market
  - Variable: *c(i,j)-j*
    - Description: "a product"
    - Dimension: 1 (i.e., no dimension)
    - Quiddity: product
- Model 2
  - Variable: d_{ij}
    - Description: "Cost of purchasing product j during month i"
    - Dimension: currency
    - Quiddity: *cost(purchase(arg(d(i,j)-i),arg(d(i,j)-j)))*
    - Quiddity Paraphrase: "the cost of purchasing a given product during a given month"
  - Variable: *d(i,j)-i*
    - Description: "a month"
    - Dimension: time
    - Quiddity: month
  - Variable: *d(i,j)-j*
    - Description: "a product"
    - Dimension: 1 (i.e., no dimension)

- $*$ Quiddity: product

The functor, *arg,* of arity 1 is being used to indicate an argument to the modeling variable. Such an argument is itself a modeling variable and has its own quiddity and dimension. The quiddity expressions are here sensitive to the difference between a time (month) and place (market), so the chance is reduced that a type 1 error will occur, providing we employ an adequately rich notion of quiddity equivalence when we apply rule 3. Quiddity expressions with arguments are only equivalent if the quiddities of their corresponding arguments are equivalent. In the present example, they are not, for the quiddities of *c(i,j)-i* and *d(i,j)-i* are not equivalent. Thus, no unique names violation has occurred and sufficient information is present to prevent the type 1 error of indicating a violation when none is present.

More complicated conditions and expressions can arise. We will consider some of them after presenting more formally and carefully, in §3, the basics of our general solution.

3 Proposed Solution

The central concept in our strategy is explicitly to represent information about model variables in the model management system. If the represented information is sufficiently rich, it can be used effectively to alert model builders of possible unique names violations, with minimal risk of type 1 and 2 errors.

For the purpose of handling unique names violations we represent two sorts of essential information about each variable: its *dimension* and its *quiddity.* We expect the concept of dimension to be a familiar one to the reader. The concept of the quiddity of a variable—which we are introducing here—is that of what it is the variable is about. We will use quiddity expressions to represent information (inevitably incomplete information) about the intended meaning of the variables in question. By adding such information we can reduce the risk of type 1 errors. For example, the dimensions of production cost and distribution cost are the same: currency. We can, however, express such difference between the two variables by using quiddity expressions, as discussed in what follows.

We note that both dimensions and quiddities should be viewed logically as terms, or referring expressions. Further, we can represent relations among these terms with statements, as in §5. We shall now describe our treatment of each of the two kinds of terms.

3.1 Dimensional Information

Physics recognizes three fundamental dimensions (among others): length, mass, and time. To these we see a need to add a fourth, currency. We also need a place holder (represented by 1) for dimensionless quantities, such as percentages.

Finally, we allow derived dimensions, such as *volume* (*length*^{3}), *acceleration, weight,* and *power.* Given these *basic dimensions* (defined for present purposes as members of the set {*length, mass, time, currency, 1, volume, acceleration, weight, power*}), we can begin to develop a particular language by stating precisely what the valid dimensional expressions are. Those for the language we will use for illustration are as follows.

1. If δ is a basic dimension, then δ is a valid dimensional expression.
2. $(\delta \cdot \gamma)$ is a valid dimensional expression if both δ and γ are valid dimensional expressions.
3. (δ/γ) is a valid dimensional expression if both δ and γ are valid dimensional expressions.
4. δ^n is a valid dimensional expression if δ is a valid dimensional expression and if either n is an integer or δ is dimensionless or each fundamental (basic and nonderived) dimension in δ has a power that is a multiple of n. In all cases, n must be dimensionless.
5. Nothing else is a valid dimensional expression.

Clearly, however, we are concerned with the family of languages generated by the different variations on such a basic vocabulary. Terms may be added or removed; it is the overall framework with which we are mainly interested.

We note as an aside that by policy we choose to use the most abstract expression possible for dimensions, e.g., we prefer to use *currency* rather than *dollars.* It is our view that complete dimensional information is best captured by use of three terms: dimension, unit, and scale. (Some authors use the term quantity as we are here using the term dimension.) Dollars and pesos are both units for the general dimension of currency. Scaling information is needed because often variables are expressed, e.g., in thousands of dollars. In general, if two variables have the same dimension, then—given their units and scales, plus general laws about the conversion relations between them—it is always possible to calculate if in fact their values are equivalent [3]. Our motivation for the policy of using only dimensional (rather than unit and scale) information about a variable *for the sake of the tomato-tomahto problem*[4] is to reduce type 2 errors. Suppose you say *tomatoes* are measured in bushels, I say *tomahtoes* are measured in quarts. We have a unique names violation, but our rules will not detect it. Both bushels and quarts are measures of volume, however, so if we state the dimensional information more generally, the problem will be detected. Alternatively, we might state the dimensional information more specifically but use laws (bushels and quarts are convertibly equivalent) to detect unique names problems. In practice, there is no reason both strategies cannot be pursued.

[4] Unit and scale information about variables is in fact present in our implementation, but is presently used for a different purpose, that of determining whether equations are dimensionally valid.

In fact, we give an example below in which we are compelled to declare unit, rather than abstract dimensional, information.

It is certainly possible further to extend the class of valid dimensional expressions by adding to the list of fundamental dimensions, but doing so is a straightforward matter and it is not to our purpose to do so here. We note that all derived dimensions should be linked by laws to fundamental dimensions. Thus, for example, we have

$$\mathit{acceleration} = \frac{\mathit{length}}{\mathit{time}^2}$$

Again, providing these laws is an easy matter and one we shall not pursue here.

Finally, we need to state certain laws for combining and manipulating dimensional expressions. These laws are well-known. We need to state the principal laws, both for the sake of facilitating implementation and for understanding the analogous laws pertaining to quiddities.

Laws for Dimensional Consistency

1. If α, β, Γ are valid dimensional expressions, with α occurring in Γ, and $\alpha = \beta$, then substituting β for α in Γ yields a valid dimensional expression. (Substitution.)
2. Two variables may be added (or subtracted) only if their dimensions are identical. (Addition.)
3. The product (quotient) of two variables having valid dimensions is dimensionally valid (Multiplication).
4. An equation is dimensionally balanced if the dimensions of its two sides are equivalent. If an equation is not dimensionally balanced, it is invalid.

Laws for Dimensional Manipulations

1. $(\alpha \cdot \beta) = (\beta \cdot \alpha)$ for any valid dimensional expressions, α, β. (Commutativity.)
2. If α, β are valid dimensional expressions, then $\alpha \cdot \frac{\beta}{\beta} = 1 \cdot \alpha = \alpha$. (Simplification.)
3. The dimension of sum (or difference) of two variables is that of either of the two variables. (Addition.)
4. The dimension of the product (quotient) of two variables is the product (quotient) of the dimensions of the two variables. (Multiplication.)

These are the main laws we will need. We will leave certain other laws (e.g., for manipulation of exponents) unstated because they are obvious.

3.2 Quiddities

We distinguish *stuff, types of stuff, attributes of stuff, types of attributes of stuff,* and *metafunctions.* Stuff includes cars, trucks, options, securities—individual things and collections of individual things. In ordinary language, stuff is usually indicated with a noun. The types of stuff concept captures adjectival information. With this concept we can distinguish, e.g., a truck tire (stuff: tire; type: truck) from a tire truck (stuff: truck; type: tire). Type information in general, and stuff type in particular, is used to answer the question What sort of stuff (attribute) is it? It's a truck. What sort of truck? A red truck. Thus, red is a stuff type.[5]

Stuff attributes represent information about some facet of the stuff in question. Attribute information is used to answer the question What is it about X that you are interested in? What is it about the truck that you are interested in? The cost or the weight? The cost and weight are stuff attributes. Types of stuff attributes, like types of stuff, answer sortal questions. What sort of cost? Materials cost. Then materials is a stuff attribute type. Finally, metafunctions capture information about the variable associated with the quiddity. For example, if x and y are variables for the price of gasoline, but x is an average price and y not, then no unique names violation should be indicated.[6]

We begin, then, to specify valid quiddity expressions by providing a basic vocabulary for each of these five categories. Again, we are here developing a specific vocabulary for the sake of illustration. Obviously, most of what we have to say applies to the family of languages generated by altering this specific vocabulary.

[5] We note that ambiguity is possible when there are iterated types. For example, we have a big red truck. Should we represent its quiddity as *big(red(truck))* or as *red(big(truck))*? English, too, is ambiguous on such matters and often relies on emphasis. Both "a *big* red truck" and "a *red* big truck" are correct. Moreover, they mean (or are used for) slightly different things. It must be accepted that any modeling exercise is also an exercise in compromise and approximation. We propose simply to ignore such distinctions and, where ambiguity—or choice—is possible, stipulate that the terms be ordered lexicographically. At the end of the day, we have *big(red(truck)),* no matter what precisely was intended. See [13] for an interesting discussion of some of the deeper issues raised by this sort of ambiguity in natural language.

[6] Our framework—with stuff, stuff types, stuff attributes, stuff attribute types, and metafunctions—bears a superficial resemblance to the entity-attribute framework, which is commonly used in data modeling. The entity-attribute framework, however, cannot (at least in our judgment) properly handle the logical complexity that can be represented in our framework. To illustrate, the analog of an entity in our framework would be a stuff type–basic stuff expression, e.g., *(diesel(fuel(arg(2)))),* but such an expression is complex; while it is a referring expression, it is logically a function, rather than a name, as entity expressions are. Again, the analog of an attribute in our system would be a stuff attribute–stuff attribute type expression, e.g., *(cost(consumption(α))),* where α is a stuff type–basic stuff expression. Here, too, we have significant logical complexity that matters to our system and that cannot be captured as a simple attribute in the entity-attribute model. Finally, there is no plausible analog in the entity-attribute model to our metafunctions.

- **basic stuff:** *car; truck; ship; index; vessel; engine; path(arg(α), arg(β)),* if α, β are declared modeling variable indicators. (See below for illustration.)
- **stuff types:** *hydrofoil; truck; ship; length.* (Note: *ship* may occur either as stuff or as a stuff type.)
- **stuff attributes:** *cost, size, availability, power, consumption*
- **stuff attribute types:** *materials, production, cost, labor, mains, auxiliaries*
- **metafunctions:** *average, max, min, sum, standard-deviation, variance, ln, exp*

We note that—in this language—all the expressions, other than stuff expressions, are functions of arity 1. Stuff expressions may have no arguments, e.g., car, ship, truck, or they may have one or more arguments, e.g., *path* has two arguments, used for indicating the endpoints. There is no specific limit on the number of arguments that a stuff term may have, other than that it be finite. We note as well that stuff terms, e.g., ship, may also be used as stuff types. This allows us to distinguish, e.g., a ship engine from a truck engine.

Given an enumeration of the vocabularies for stuff, stuff types, stuff attributes, stuff attribute types, and metafunctions, we can state precisely what the valid quiddity expressions are. We begin by defining a valid stuff term. These definitions will shortly be illustrated with some examples.

1. If α is in the vocabulary of basic stuff expressions, then α is a valid stuff term, providing that each of its arguments has the form *arg(n)*, where n is an integer identified with a declared variable (or is a declared variable-indicating expression).
2. If α is in the vocabulary of basic stuff expressions, then $\alpha[arg(n)]$ is a valid stuff term, where $\alpha[arg(n)]$ has one more argument than α and n is an integer identified with a declared variable (or is a declared variable-indicating expression) with a quiddity of *index.*
3. $\phi(\alpha)$ is a valid stuff term if α is a valid stuff term and ϕ is in the vocabulary of stuff types.
4. $\phi(\alpha)$ is a valid stuff term if α is a valid stuff term and ϕ is in the vocabulary of metafunctions.
5. Nothing else is a valid stuff term.

Given this, the valid quiddity terms may be easily characterized.

1. If α a valid stuff term, then α is a valid quiddity term.

2. $\phi(\alpha)$ is a valid quiddity term if α is a valid stuff term, and ϕ is in the vocabulary of stuff attributes.
3. $\phi(\alpha)$ is a valid quiddity term if α is a valid quiddity term and ϕ is in the vocabulary of metafunctions.
4. $\phi(\alpha)$ is a valid quiddity term if α is a valid quiddity term and ϕ is in the vocabulary of stuff attribute types.
5. $\alpha \cdot \beta$ and α/β are valid quiddity terms if α and β are valid quiddity terms.
6. Nothing else is a valid quiddity term.

Some examples will help to communicate the sense and import of these definitions. Consider the following equational model for estimating the total fuel costs in a given year, t, for a vessel:

$$yfc_t = fCost_t \cdot (fConsM + fConsA) \tag{4}$$

The four variables in this model, 4, have the following intended interpretations:

yfc_t	fuel cost for a vessel during year t, in dollars
$fCost_t$	cost of fuel in dollars per gallon in year t
$fConsM$	fuel consumed by Mains
$fConsA$	fuel consumed by Auxiliaries

This is the top-level equation for a U.S. Coast Guard model, called CAPS ("comparing alternative propulsion systems") [5, 6]. Each of the three right-hand-side variables in CAPS (1) is itself the left-hand side of a submodel. For the moment, however, we shall focus on (1).

To begin, note the associated dimensions for these four variables:

yfc_t	$\mathit{currency}$
$fCost_t$	$\frac{\mathit{currency}}{\mathit{volume}}$
$fConsM$	volume
$fConsA$	volume

Note further that, using the rules of combination and manipulation described above for dimensions, the two sides of the CAPS equation are dimensionally identical, as they should be.

What are the quiddities of these variables? We begin with yfc_t, which is about (has quiddity covering) a vessel during a particular year, or time. The basic stuff for this variable is a vessel at a time. We do not have stuff types in the present case, but we do have stuff attributes. What is it about the vessel that we are interested in? Its fuel. What kind of fuel? Diesel fuel. What about the diesel fuel? We are interested in the vessel's consumption of fuel. What about the consumption? The cost. This leads to the following quiddity expression for yfc_t:

$$\mathit{cost}(\mathit{consumption}(\mathit{diesel}(\mathit{fuel}(\mathit{vessel}(\mathit{arg}(\mathit{1})))))) \tag{5}$$

Since *vessel* has an argument, we must declare a quiddity and dimension for the argument. *arg(1)* (or t in yfc_t) is an index variable. Its stuff is *index* and its dimension is *time.*

$fCost_t$ is about the cost of diesel fuel in year t, regardless of how the fuel is to be consumed. Its quiddity, then, is:

$$cost(diesel(fuel(arg(2)))) \tag{6}$$

Since *fuel* has an argument, we must declare a quiddity and dimension for the argument. *arg(2)* (or t in $fCost_t$) is also an index variable. Its stuff is *index* and its dimension is *time.*

$fConsM$ is about the consumption of diesel fuel by a vessel, due to the use of the main propulsion plant in the vessel. This yields quiddity:

$$mains(consumption(diesel(fuel(vessel)))) \tag{7}$$

$fConsA$, similarly, is about the consumption of diesel fuel by a vessel, due to the use of the auxiliary propulsion plant in the vessel. This yields quiddity:

$$auxiliaries(consumption(diesel(fuel(vessel)))) \tag{8}$$

3.3 CAPS Example, Continued

The Coast Guard's CAPS model is, as we have seen, composed of three submodels, which are used to determine the values of the variables $fCost_t$, $fConsM$ and $fConsA$. The full CAPS model—its top-level equation and the three equations for its submodels—are as follows.

CAPS:

$$yfc_t = fCost_t \cdot (fConsM + fConsA) \tag{9}$$

fuel_COST:

$$fCost_t = c0 \cdot (1 + swag1 \cdot t - swag2 \cdot t^2) \tag{10}$$

fuel_CONS_M:

$$fConsM = (\sum_{i=1}^{4} fracS_i \cdot hpreq_i \cdot sfcS_i) \cdot ophrsM/gamma1 \tag{11}$$

fuel_CONS_A:

$$fConsA = \frac{(\sum_{i=1}^{3} fracL_i \cdot kwreq_i \cdot sfcL_i) \cdot ophrsA}{gamma1 \cdot gamma2} \tag{12}$$

We note that two of the submodels, 11 and 12, contain a common parameter, *gamma1,* which might—should the two models name it differently—be the object of a unique names violation, as may be the various indices on the variables.

The variables and their intended interpretations for the full, integrated model are as follows.

t: time index.
i: speed code for vessel.
j: load code for vessel.
yfc_t: fuel cost for year t.
fCost_t: unit cost of fuel in the year t.
fCons: amount of diesel fuel consumed per year.
fConsM: amount of fuel consumed by Mains.
fConsA: amount of fuel consumed by Auxiliaries.
ophrsM: number of mains operating hours per year.
ophrsA: number of auxiliaries operating hours per year.
fracS_i: fraction of time at speed i.
fracL_i: fraction of time at load i.
hpreq_i: horse power required at speed i.
sfcS_i: specific fuel consumption of diesel engine (lb / hp-hr) at vessel speed i.
kwreq_i: electrical power required at load i.
sfcL_i: specific fuel consumption of diesel engine (lb / hp-hr) at electrical load i.
gamma1: pounds per gallon of diesel fuel.
gamma2: kilowatts per horse power.
swag1: time-dependent expansion factor.
swag2: time-dependent expansion factor.
c0: unit cost of fuel in year 0 (now).

We now give quiddities for each of these variables that have not had their quiddities defined above. For the sake of simplicity, we will make free use of derived dimensions, e.g., weight and velocity.

variable	*quiddity*	*dimension*
i	*index*	*velocity*
j	*index*	*power*
ophrsM	*mains(vessel)*	*time*
ophrsA	*auxiliariess(vessel)*	*time*
$fracS_i$	*percentage(arg(i))*	1
$fracL_j$	*percentage(arg(j))*	1
$hpreq_i$	*mains(consumption(vessel(arg(i)))*	power
$kwreq_j$	*electricity(consumption(vessel(arg(j)))*	power
$sfcS_i$	$\frac{mains(consumption(diesel(fuel(vessel(arg(i)))))}{mains(vessel)\cdot mains(consumption(vessel))}$	$\frac{weight}{time\cdot power}$
$sfcL_j$	$\frac{mains(consumption(diesel(fuel(vessel(arg(j)))))}{mains(vessel)\cdot mains(consumption(vessel))}$	$\frac{weight}{time\cdot power}$
gamma1	1	$\frac{weight}{volume}$
gamma2	1	$power/power$
swag1	1	$1/time$
swag2	1	$1/time$
c0	$cost(diesel(fuel))$	$currency/volume$

Briefly, anticipating the discussion in §5, we note that the quiddities in equation 11 balance. After the "summing out" of the subscript (or index) variable i, the quiddity for the summation in 11 is:

$$\frac{mains(consumption(vessel)) \cdot mains(consumption(diesel(fuel(vessel))}{mains(vessel) \cdot mains(consumption(vessel))} \tag{13}$$

Given this, and elementary manipulations (principally cancellations), equation 11 is brought into balance with regard to quiddity. Equations 10 and 12 are either very simple, or are similar to 11 with respect to quiddity balancing.

4 Implementation

In this section we discuss a prototype implementation, in the model management system TEFA, of our proposed solution for the unique names violation problem. TEFA is currently being developed and is a main part of a general–purpose DSS shell, MAX, in use at the U.S. Coast Guard's R. & D. Center and its Office of Acquisition. This system is implemented in Prolog and runs on Macintosh computers. The implementation of TEFA is based on the embedding of an executable modeling language $L_\downarrow$ in another formal model management language called $L_\uparrow$. The embedded languages technique is described in more detail in [6, 7]. The language $L_\downarrow$ has the ability to represent mathematical and qualitative information related to models, while $L_\uparrow$ is used to specify $L_\downarrow$ (and other languages) and to express this information by adding sentences in these languages. TEFA aims to support modelers in the various phases in the modeling life cycle, including model formulation, model validation, model execution and solution, explanation of solution, and reporting, for certain classes of mathematical models (mainly hierarchical systems of equations, possibly with conditions; mathematical programming models are also representable in TEFA, but this is not supported in the currently-delivered version). Although TEFA does not yet provide support functions for all of the modeling life cycle, various useful model management tasks are implemented in $L_{\uparrow}$. The one we are concerned with in this section is the model validation function.

We wish to ensure that models defined in TEFA, including models that combine several other models, are valid. A necessary condition for that is that there be no unique names violations (UNVs) in the integrated or super–model. We will discuss our implementation for the detection of UNVs based on the rules presented in the previous section. We are interested in computing the function *tomahto*,

$$tomahto : \Delta_M \mapsto \Delta_V \times \Delta_V$$

where Δ_M is a set of models and Δ_V is a set of variables defined in the modeling language. Given a model M, this function determines pairs of variables that appear to constitute a UNV in the integration of M with the submodels associated with M. (A submodel of M is a model *called* by M to compute a variable used in M.) With no loss of generality, we make the assumption here that there is no UNV *within* a model, i.e., no two syntactically distinct variables within the same model refer to the same thing. This is a reasonable assumption since our problem is that UNVs occur in the *integration* of several models. Our program for computing the tomahto function is informally described below.

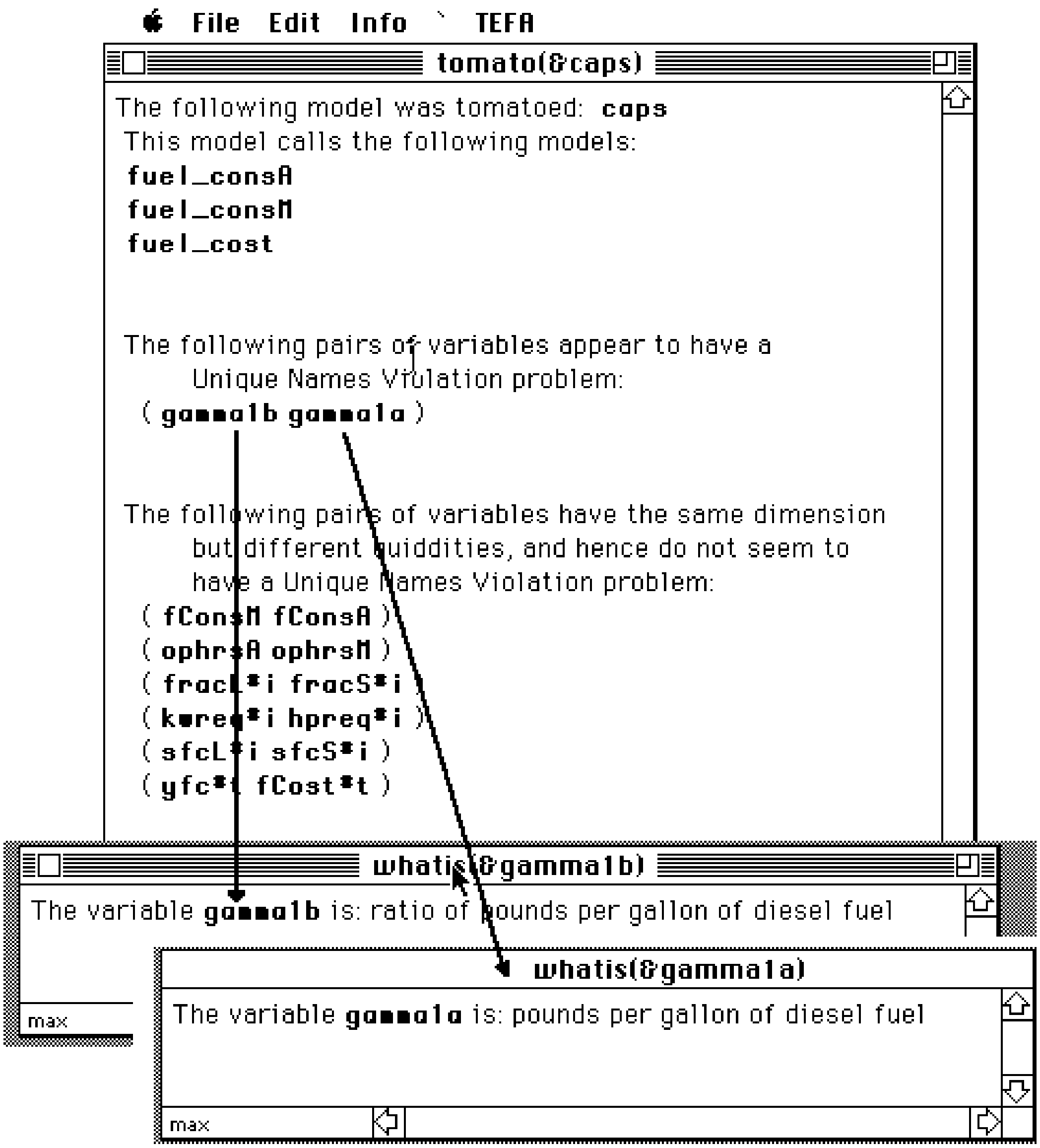
1. Given a model M, let M' be the set of models, including M, that are submodels of M. This set is obtained by a TEFA function that determines the models called by M. Denote by UNV_M the set of possible UNVs in combining M and its submodels.
2. For every pair (M_1, M_2) of models in M', perform step 3.
3. Let Var_{M_1} and Var_{M_2} be the sets of variable names occurring in all the expressions associated with M_1 and M_2 respectively. These sets are computed by a TEFA meta-interpreter that interprets the expressions associated with a model. For each pair (V_1, V_2) in the Cartesian product of these sets, perform step 4.
4. UNV detection: Given a pair of variables (V_1, V_2) determine the dimensions D_1 and D_2, and quiddities Q_1 and Q_2 of these variables. These are obtained from declarations about the variables. If D_1 is equivalent (under dimensional transformation) to D_2 *and* Q_1 is equivalent (under quiddity transformation) to Q_2, then (V_1, V_2) consitute a possible UNV.[7] Enter (V_1, V_2) in the set UNV_M.

[7] Computationally, it is more efficient to perform this operation in the following two steps: 1) Determine dimensions of V_1 and V_2. If they are unequal, reject this pair, else perform step 2, 2) Determine quiddities of V_1 and V_2. If they are equivalent, we have a possible UNV. This is in fact how our implementation works.

TEFA interacts with the user interface system of MAX via a formal language for obtaining commands and other inputs through the interface, and for presenting outputs such as reports to the user. TEFA realizes the concept of *generalized hypertext* [5, 8, 9] by generating and identifying hypertext *nodes* within these reports, and by performing a variety of operations to follow the *links* for the nodes. As we discuss below, this feature is of material help in the detection and correction of UNVs.

We intend the detection and correction of UNVs to be a joint effort between the system and the modeler, with the machine doing most of the detection of *possible* UNVs, and the modeler confirming UNVs and correcting them. Hence it is important that the system be able to provide easy and quick access to a rich variety of information about the models and the variables, so that the modeler can easily make informed decisions. Since the detection of UNVs involves pairwise comparisons of variables in every pair of models to be combined, it is easy to see that there is a massive amount of information that is relevant to the confirmation and correction of UNVs. It is therefore necessary for the machine to present this information in manageable and meaningful chunks, and to provide quick and easy access to the rest. This is made possible in our implementation due to the generalized hypertext features in TEFA. We illustrate this using the CAPS model. The presentation of information in various windows on the screen, and the ability to point and click at various hypertext nodes is managed by MAX's user interface for generalized hypertext.

We now illustrate the UNV detection feature in TEFA. For the purpose of this discussion let us assume that the fuel-cons-A and fuel-cons-M models were built by two separate modelers, who used variable names *gamma1a* and *gamma1b* for the variable *gamma1* above. Thus there is a UNV in the CAPS model, which calls these two sub-models. The top section of the following figure depicts the report produced by TEFA on being asked by the user to 'tomato' the caps model, i.e., to detect UNVs. This report indicates a pair of variables causing a UNV, and the user points and clicks on the variable names (which are hypertext buttons) to get a brief description about these two variables, as indicated by the arrows connecting these variable names to the descriptions.



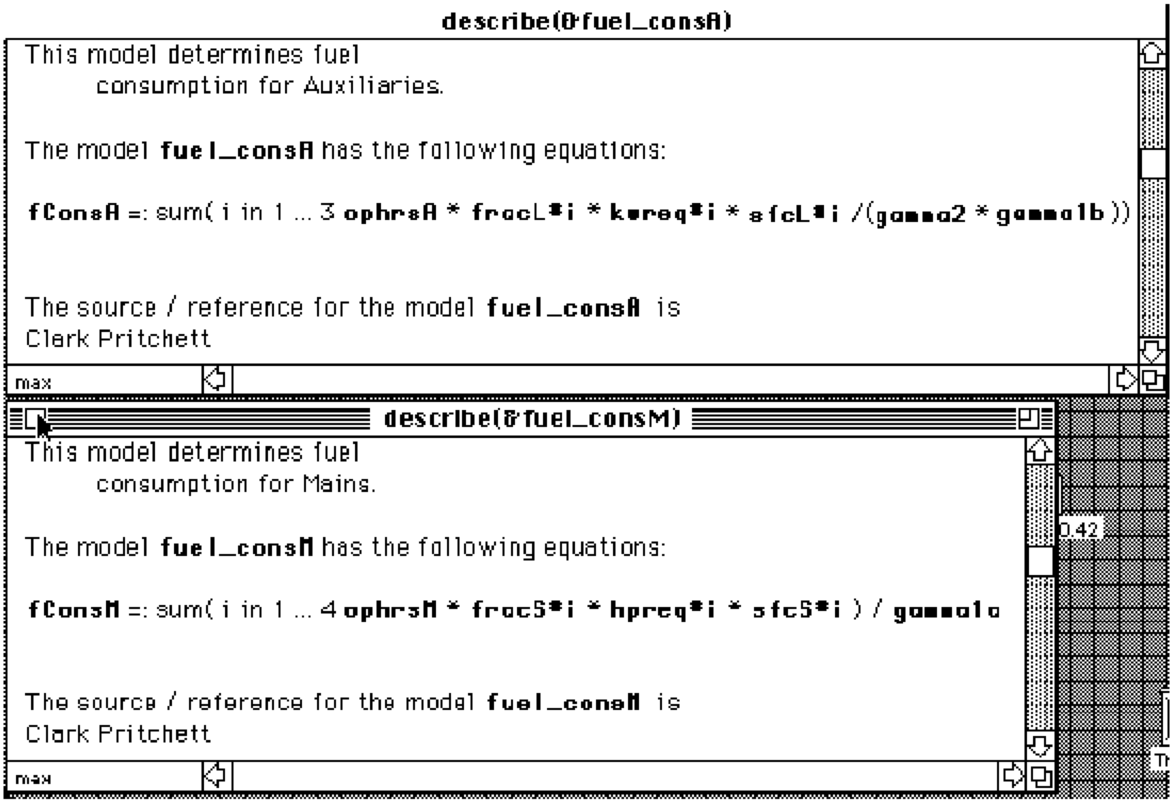
```
The following model was tomatoed:  caps
 This model calls the following models:
 fuel_consA
 fuel_consM
 fuel_cost


 The following pairs of variables appear to have a
        Unique Names Violation problem:
  ( gamma1b gamma1a )


 The following pairs of variables have the same dimension
        but different quiddities, and hence do not seem to
        have a Unique Names Violation problem:
  ( fConsM fConsA )
  ( ophrsA ophrsM )
  ( fracL*i fracS*i )
  ( kwreq*i hpreq*i )
  ( sfcL*i sfcS*i )
  ( yfc*t fCost*t )
```

```
The variable gamma1b is: ratio of pounds per gallon of diesel fuel
```

```
The variable gamma1a is: pounds per gallon of diesel fuel
```

Figure 1: Detecting UNVs.

On obtaining these descriptions the user can easily ascertain that these two variables indeed refer to the same thing and must therefore have the same name. The following figure shows information about the fuel-cons-A and fuel-cons-m models provided by TEFA, which shows that the variable *gamma1* has different names in these models.



describe(&fuel_consA)

```
This model determines fuel
      consumption for Auxiliaries.

The model fuel_consA has the following equations:

fConsA =: sum( i in 1 ... 3 ophrsA * fracL*i * kwreq*i * sfcL*i /(gamma2 * gamma1b ))


The source / reference for the model fuel_consA is
Clark Pritchett
```

describe(&fuel_consM)

```
This model determines fuel
      consumption for Mains.

The model fuel_consM has the following equations:

fConsM =: sum( i in 1 ... 4 ophrsM * fracS*i * hpreq*i * sfcS*i ) / gamma1a


The source / reference for the model fuel_consM is
Clark Pritchett
```

Figure 2: Examining UNVs.

5 Rules for Quiddity Manipulation

Just as there are laws for manipulation of dimensional expressions, see §3.1 above, so there are laws for manipulation of quiddity expressions. Moreover, it is our view that equations should balance with respect to quiddities, just as they should with respect to dimensions (and for the same reason: for the sake of coherence). Returning to the CAPS example and its assigned quiddities, the problem is to understand how to combine the quiddities so that the equations balance with respect to quiddities. The situation is more complex than that for dimensions. We shall proceed informally for the present. Although we shall be more rigorous in the sequel, because the main subject of this paper is automated assistance for the problem of unique names violations in model integration, we will not provide more than some rough details for the claim that quiddities can be combined in a fashion like that for dimensions. Our purpose here is to illustrate what is involved in combining quiddities and to make it plausible that it can be done in a rigorous and principled way.

First, consider the quiddity of $(fConsM + fConsA)$ (recall expressions 7 and 8, above, at the end of §3.2). *mains* and *auxiliaries* are both stuff attribute *types.* They are both special cases of the more general quiddity, *consumption(diesel(fuel(vessel))).* As such, it makes sense to add them providing that the resulting quiddity just is that more general one. After all, apples are a kind of fruit, as are oranges. Adding apples and oranges gives us fruit, or expressed as quiddities:

$$apple(fruit) + orange(fruit) = fruit \tag{14}$$

The problem of combining quiddities for the rest of $fCost_t \cdot (fConsM + fConsA)$ is more challenging. The problem is to handle in a principled and plausible way the product of two quiddity terms: $\mathit{consumption}(\mathit{diesel}(fuel(\mathit{vessel})))$ and $\mathit{cost}(\mathit{diesel}(\mathit{fuel}(\mathit{arg}(\mathit{2}))))$, call them α and β. Beginning on the inside of the quiddity terms, note that the two expressions seem to be about different things—fuel and vessel—and one is indexed, while the other is not. Consider the former issue first. Note that inside, α is more specific than β, for α is about diesel fuel *for a vessel,* while β is about (any old) diesel fuel. But if β is about diesel fuel generally, then it is also about diesel fuel for vessels. Thus we can transform β to β':

$$\mathit{cost}(\mathit{diesel}(\mathit{fuel}(\mathit{vessel}(\mathit{arg}(\mathit{2}))))) \qquad \beta'$$

We now have the problem of reconciling *vessel* and *vessel(arg(2)).* We do so simply by transforming the former to the latter, using the same justification just employed. We opt for the more specific case, since the general term—here, *vessel*—encompasses the more specific term—here, *vessel(arg(2)).* We now must reconcile β' and α':

$$\mathit{consumption}(\mathit{diesel}(\mathit{fuel}(\mathit{vessel}(\mathit{arg}(\mathit{2})))) \qquad \alpha'$$

The remaining problem of reconciling α' and β' is straightforward. Neither is a special case of the other, nor are they special cases of some third thing (as were apples and oranges of fruit). Consequently, we must combine α' and β' in some way. Clearly, we have two choices: the combination is either a consumption-cost or a cost-of-consumption. We note that "cost" appears, above, in our basic quiddity vocabulary both as a stuff attribute and as a stuff attribute type, while "consumption" appears only as a stuff attribute. Thus, our choice is clear. The resulting quiddity for the right-hand side of the CAPS equation is γ:

$$\mathit{cost}(\mathit{consumption}(\mathit{diesel}(\mathit{fuel}(\mathit{vessel}(\mathit{arg}(2))))) \qquad \gamma$$

Further, since *arg(1)* and *arg(2)* have common stuff and dimensions, we can now say that the CAPS equation is balanced with regard to quiddities.[8]

Consider the situation more broadly. Dimensions and quiddities have similar rules for their manipulation. We listed several of these rules for dimensions in §3.1. Our purpose now is to present some analogous rules for quiddities.

Generalizing from our earlier discussion, the essential concepts for quiddity manipulation are the notions of a *quiddity case* and *quiddity construct.* Variables may be added only if they have identical dimensions. On the quiddity side, variables may be added only if they have identical quiddity cases. Unlike as for dimensions, however, variables may have more than one quiddity case. Recall our example of adding apples and oranges, expression 14. Apples and oranges may be added because their quiddities are both cases of fruit; they have identical quiddity cases. We can state rules for identifying identical quiddity cases. We write $\alpha \overset{\text{quidcase}}{=} \beta$ to say that quiddity expressions α and β have identical quiddity cases. The following rules generalize our previous discussion.

1. If $\alpha = \beta$ then $\alpha \overset{\text{quidcase}}{=} \beta$ with case α.
2. $\alpha(\beta) \overset{\text{quidcase}}{=} \delta(\gamma)$ with case ρ, if $\beta \overset{\text{quidcase}}{=} \gamma$ with case ρ.

(Rule 2 allowed us to get the quiddity of $(fConsM + fConsA)$ from 7 and 8, yielding: *consumption(diesel(fuel(vessel))).*)

When two variables are multiplied (divided) the dimension of the resulting expression is the multiplication (division) of the dimensions of the two variables. On the quiddity side, when two variables are multiplied (divided) the quiddity of the resulting expression is a quiddity construct. Again, we can state rules for identifying identical quiddity constructs. We write $\alpha \overset{\text{quidcons}}{=} \beta$ to say that quiddity expressions α and β have identical quiddity constructs. The following rules generalize our previous discussion.

1. $\alpha(\beta(\gamma)) \overset{\text{quidcons}}{=} \alpha(\delta) \cdot \beta(\epsilon)$ with construct $\alpha(\beta(\gamma))$ if α is a (stuff or stuff attribute) type and β is not, and $\gamma \overset{\text{quidcons}}{=} \delta \overset{\text{quidcons}}{=} \epsilon$
2. $\alpha(\beta) \overset{\text{quidcons}}{=} \alpha(arg(\gamma))$ with construct $\alpha(\beta(\text{arg}(\gamma)))$.

[8] Of course, had "consumption" also appeared as a stuff attribute type, we would have had an ambiguity, but then it would have been resolved lexicographically and the CAPS equation would still end up being in quiddity balance.

(The moves from α to α', and from β to β' are sanctioned by rule 2. The move from α' and β' to γ is sanctioned by rule 1.)

Clearly, for both quiddity cases and quiddity constructs more elaborate rule sets are possible. Adding rules, while it reduces the risk of type 1 error, increases computational cost, so that a tradeoff must be made in any implementation.

With these rules at hand, we can state the quiddity version of the laws for manipulation of expressions.

Laws for Quiddity Manipulations

1. If α, β, Γ are valid quiddity expressions, with α occurring in Γ, and $\alpha = \beta$, then substituting β for α in Γ yields a valid quiddity expression. (Substitution.)
2. $(\alpha \cdot \beta) = (\beta \cdot \alpha)$ for any valid quiddity expressions, α, β. (Commutativity.)
3. If α, β are valid quiddity expressions, then $\alpha \cdot \frac{\beta}{\beta} = 1 \cdot \alpha = \alpha$. (Simplification.)
4. Two variables may be added (or subtracted) only if their quiddities have identical cases, and the resulting quiddity is their most specific common quiddity case. (Addition.)
5. The quiddity of the product (quotient) of two variables is the product (quotient) of the quiddities of the two variables, after finding identical constructs. (Multiplication.)
6. An equation is balanced with respect to quiddity if the quiddities of its two sides are equivalent, i.e., are identical or have identical cases or constructs. If an equation is not balanced with respect to quiddity, it is invalid.

This will suffice as an introduction to quiddity manipulations. Much more remains to be learned about how to do it and what can be done with such manipulations.

6 Discussion

What we have proposed here is an instance of a more general move, that of declaring information *about* variables (and other model elements) and of exploiting these declarations inferentially in order to support model management functions. This is a particularly apt strategy when the underlying approach to model management is—as is the case with TEFA—that of an executable modeling language [15, 10, 17]. Indeed, the ready applicability of this move, and the generality of its usefulness, constitutes in our view yet another reason to prefer an executable modeling language approach to model management generally (and an embedded languages approach in particular [4, 6, 7]). As noted by Bradley and Clemence [10] (our notion of quiddity is a generalization of their notion of concept), the apparatus employed in declaring and exploiting information about model elements is pretty much independent of the particular executable model language. Further, although a particular problem (tomato-tomahto, or synonymy) occasioned our proposal, the apparatus of our solution—dimensions and quiddities, plus laws about them—is useful on other problems as well. A simple modification to the rules described in §4 will provide a means of dealing with the homonym (tomato-tomato) problem. Further, also noted by Bradley and Clemence, the existence of dimensional declarations can be exploited for various model validation tests, since equations must balance dimensionally (and, we would add, with respect to quiddity).

Given this generality and general utility, the work described here is only a beginning. We see significant opportunities for future work. In particular, the rules for quiddity manipulation discussed in §5 are quite elementary. Much richer rule sets and inferences could be applied. Several kinds of such rule sets are particularly worth investigating. First, more extensive use of classification (normally, "isa") hierarchies could be used for determining quidcase equivalences. There is much more knowledge to exploit here than the fact that apples and oranges are both fruits. Second, rules for further reducing the risk of type 1 and 2 errors could be added. Suppose, for example, that two variables have the same quiddity and dimension, but both have their values determined by models. In principle it is possible, and for simple models it could be practicable, automatically to check the two models for equivalence. Third, we introduced metafunctions as part of our vocabulary, in §3.2, for expressing quiddities, but we have given no rules for exploiting them. This can certainly be done and there are several uses of such information. For example, suppose two variables are equivalent, with respect to dimension and quiddity, up to a simple transform, e.g. log, squaring, polarity of a 0-1 indicator variable, and so forth. Given a list of transforms, it is possible automatically to check whether two seemingly distinct variables are in fact equivalent under transformation. If you say *tomato* and I say $e^{tomahto}$, then we still have a problem.

With more experience on actual modeling situations and with further application of creative thought, no doubt many other opportunities will be found to extend and enhance the apparatus we have described.

References

[1] Batini, C. and M. Lenserini, "A Methodology for Data Schema Integration in the Entity Relationship Model," *IEEE Transactions on Software Engineering,* **SE-10**, 6, (1984), 650-663.

[2] Batini, C., M. Lenserini, and S. Navathe, "A Comparative Analysis of Methodologies for Database Schema Integration," *ACM Computing Surveys,* (1987).

[3] Bhargava, Hemant K., "A Simple and Fast Numerical Method for Dimensional Arithmetic," Naval Postgraduate School, working paper number 90-01, March 1990.

[4] Bhargava, Hemant K., *A Logic Model for Model Management: An Embedded Languages Approach,* University of Pennsylvania, Ph.D. dissertation, (Decision Sciences Department, working paper 90-09-01), 1990.

[5] Bhargava, Hemant K., Michael P. Bieber, and Steven O. Kimbrough, "Oona, Max, and the WYWWYWI Principle: Generalized Hypertext and Model Management in a Symbolic Programming Environment," *Proceedings of the Ninth International Conference on Information Systems,* Janice I. DeGross and Margrethe H. Olson, eds., (November 30-December 3, 1988), 179-191.

[6] Bhargava, Hemant K. and Steven O. Kimbrough, "On Embedded Languages for Model Management," *Proceedings of the Twenty-Third Annual Hawaii International Conference on System Sciences, Volume III,* Jay F. Nunamaker, ed., IEEE Computer Society Press, Los Alamitos, California, 1990, pp. 443-452; revised and expanded as [7].

[7] Bhargava, Hemant K. and Steven O. Kimbrough, "Model Management: An Embedded Languages Approach," University of Pennsylvania, Decision Sciences Department, working paper, 1990.

[8] Bieber, Michael P. and Steven O. Kimbrough, "On Generalizing the Concept of Hypertext," University of Pennsylvania, Department of Decision Sciences, working paper, 1989.

[9] Bieber, Michael P. and Steven O. Kimbrough, "Towards a Logic Model for Generalized Hypertext," University of Pennsylvania, Department of Decision Sciences, working paper, 1989.

[10] Bradley, G.H. and R.D. Clemence, Jr., "Model Integration with a Typed Executable Modeling Language," *Proceedings of the Twenty-First Hawaii International Conference on System Sciences, Vol. III,* (January 1988), 403-410.

[11] Casonova, M. and M. Vidal, "Towards a Sound View Integration Methodology," *Proceedings of the 2nd ACM SIGACT/SIGMOD Conference on the Principles of Database Systems,* (1983), 36-47.

[12] Dayal, U. and H. Hwang, "View Definition and Generalization for Database Integration in a Multidatabase System," *IEEE Transactions on Software Engineering,* **SE-10**, 6, (1984), 628-644.

[13] Dretske, Fred, "Contrastive Statements," *Philosophical Review,* **81**, (1972), 411-37.

[14] ElMasri, R., J. Larson, and S. Navathe, "Integration Algorithms for Federated Databases and Logical Database Design," Technical Report, Honeywell Corporate Research Center, 1987.

[15] Fourer, R., "Modeling Languages Versus Matrix Generators for Linear Programming," *ACM Trans. Math. Software,* **9**, 2, (June 1983), 143-83.

[16] Genesereth and Nils J. Nilsson, *Logical Foundations of Artificial Intelligence,* Morgan Kaufmann Publishers, Inc., Palo Alto, California, 1987.

[17] Geoffrion, Arthur M., "Reusing Structured Models via Model Integration," *Proceedings of the Twenty-Second Hawaii International Conference on System Sciences, Vol. III,* (January 1988), 601-611.

[18] Yao, S.B., V. Waddle and B. Housel, "View Modeling and Integration Using the Functional Data Model," *IEEE Transactions on Software Engineering,* **SE-8**, 6, 544-553.