Knowledge Representation for Commonality*

Dorian P. Yeager, Ph.D.
Department of Computer Science

The University of Alabama
Tuscaloosa, AL 35487

February 5, 1990

Abstract

Domain-specific knowledge necessary for commonality analysis falls into two general classes: commonality constraints and costing information. Notations for encoding such knowledge should be powerful and flexible and should appeal to the domain expert. The notations employed by the CAPS analysis tool are described herein. Examples are given to illustrate the main concepts.

1 Introduction

Commonality is the extent to which a system employs common designs to fulfill similar functions. A system which uses a large number of individually tailored components for specific functions thus possesses little commonality, whereas a system which maximizes the number of applications of each component is said to possess a high degree of commonality. The knowledge which allows an engineer to decide which components are able to serve in multiple applications, and thereby to eliminate unnecessary duplication of functionality, is very domain specific. Commonality knowledge capture is therefore greatly aided by a notation which appeals to the domain expert and which provides a natural way to describe commonality considerations.

With Prolog, we are able to communicate this type of information in a way which is difficult to reproduce with other types of notation. For example, to say that

*Work supported by NASA grants NGT-01-002-099 and NAG8-718.

motor A can serve in motor B's stead provided A has at least as high a power rating as B, we use the following Prolog statement:

```
can_substitute(A,B) :-
  motor(A), motor(B),
  power(A,A_power), power(B,B_power),
  A_power>=B_power
```

The above Prolog production (rule) states that "A can substitute for B provided A is a motor, B is a motor, A's power rating is A power, B's power rating is B power, and A power $\geq$ B power". Prolog requires, of course, that the predicates "motor" and "power" be defined. Defining the former requires that we name each motor with its own symbol, say m1, m2, etc. We then define the predicate "motor" with a series of declarations as follows.

```
motor(m1)
motor(m2)
motor(m3)
```

etc.

The power ratings of the various motors are communicated in a straightforward fashion, as follows.

```
power(m1,1.7)
power(m2,3.3)
power(m3,3.3)
```

etc.

Thus Prolog is a reasonable tool for communicating commonality constraints. Its drawbacks are many, however. Apart from its strange syntax, Prolog carries with it the overhead of explicitly declaring predicates like "motor" and "power" above, and using them in production rules. Also, the unification mechanism of Prolog (see Clocksin and Mellish [1]) is too costly and too general for this specific application. The knowledge to be communicated here is more along the lines of "object A is related to object B" than "statement P is true provided statement Q is true". A notation is needed which has the power of Prolog but is more familiar in form and does not carry with it the necessity to embrace a wider domain of application.

2 The CAPS Language

A tool under development for NASA, entitled "Commonality Analysis Problem Solver", or CAPS, has incorporated into its allowable input forms a notation for communicating commonality constraints. This notation was designed with the following objectives in mind: (1) it should be algebraic in nature, resembling the query languages of relational databases; and (2) it should incorporate enough of the power of the Prolog language so that the communication of a relationship between objects is easy and natural.

The nature of the knowledge to be communicated here requires that two objects be described for the purpose of relating one to the other. Therefore it is convenient to use two separate algebraic expressions. Each of these expressions is a database query which isolates a subset of the set of objects under consideration. The two general forms are as follows:

for $<expression_1>$ **allow** $<expression_2>$
for $<expression_1>$ **disallow** $<expression_2>$

The above notation either "allows" or "disallows" the substitution of objects satisfying $<expression_2>$ for objects satisfying $<expression_1>$. Thus for a database containing information on wrenches, the statement

```
for millimeters=10 allow inches=13/32
```

might be used to permit the cost analysis phase of CAPS to consider using a 13/32 inch wrench in place of a 10 millimeter wrench.

What CAPS borrows from Prolog is its view of free variables and the comparison/assignment operator. In Prolog, the "=" symbol is used both for comparison and for assignment. If both sides of the equality can be evaluated, then a true or false value will be returned in the usual way. However, if one side of the equality is a free variable, i.e. a variable which has not been bound to a value during the current firing of the current production, then the other side's value is bound to it and a value of "true" returned. Using the same hypothetical wrench database as above, consider the following statement.

```
for millimeters=x allow inches>=0.0394*x and inches<=0.042*x
```

In this example, a larger set of wrenches is considered. The expression

```
millimeters=x
```

refers to the entire database, since x is a free variable and hence can be bound to any one of the values of the "millimeters" attribute. The key here is that once x is bound in that first expression to a specific value applicable to a specific wrench, the remainder of the wrenches in the database are then examined to see if any of them satisfy the expression

```
inches>=0.0394*x and inches<=0.042*x
```

As before, any which do are accepted as substitutes for the wrench in question. The same process is repeated for each wrench in the database.

Unlike Prolog, which is case-sensitive and dictates that all variables begin with a capital letter, CAPS is not case-sensitive and allows any identifier not already bound to an attribute name, a keyword, or a value to be used as a free variable.

3 Semantic Issues

A database query is a single boolean expression which identifies a subset of the elements in the database. It is a declarative statement about the properties of the members of that subset. For example, with a relational database it would be possible to fetch the entire set of wrenches for which

```
inches > 5/16 and inches <= 7/8
```

One could be sure then that in the subset fetched all wrenches would have the stated property. The semantics of a CAPS "for" statement are not so simple, however. Each "for" statement, instead of being a declarative definition of a relation on a set of objects, is a command used to *modify* such a relation. The relation in question is the substitutability relation on a set of objects. During its analysis of a given database, CAPS initially assumes that there are no allowable substitutions except the trivial ones, in which each object "substitutes" for itself. As each "for" statement is fired, the relation is altered to achieve a cumulative effect. We say that the CAPS notation is "history sensitive" in that the effect of each "for" statement depends not only on the statement itself but also on the internal state of CAPS, and that internal state is a function of all prior CAPS commands.

The usefulness of this aspect of CAPS can be seen in the following example, involving a database containing information on storage tanks.

```
for volume=x allow volume>=x
for liquid disallow gas
for gas disallow liquid
```

The net effect of firing the three "for" statements in the sequence indicated is to allow larger tanks to substitute for smaller ones, but not to allow the substitution of tanks designed to hold liquids for those designed to hold gasses, or vice versa.

Another semantic issue is that of binding times. The binding of a free variable takes place many times during the execution of a single "for" statement. For this reason CAPS must incorporate a very flexible strategy for binding variables to values. To illustrate, consider the "define" statement, with syntax as follows:

define <*identifier*> **as** <*expression*>

Here the identifier is associated with an unevaluated expression. If free variables appear in that expression, they will take on whatever values are current when the identifier is used in some subsequent computation, such as the firing of a "for" statement. In the same way, if there are field names in the expression, they will take on the appropriate values each time the identifier is referenced. Consider, for example, the sequence of CAPS commands that follows.

```
define rel_power as power/weight
for rel_power=x allow rel_power>=x
```

Here "rel power" is defined as the ratio of power to weight. Since these are both field names, it makes no sense to evaluate rel power at the point where it is defined. Rather, the expression "power/weight" is stored internally and evaluated each time it is needed. In the example, if there are n items in the database, the expresion is evaluated n^3 times to fire the "for" statement. The wastefulness of this approach is clear, since there are in fact only n possible values for "rel power". To avoid the extra complexity, CAPS allows its user to add "rel power" as a new field in the database. The statement

```
add rel_power
```

accomplishes this, provided "rel power" has already been defined as above.

To further illustrate, we use a variant of a type of commonality analysis problem originally formulated by Thomas ([3]). Consider a database containing information on utility interface plates. Each interface plate consists of a set of connections for utilities. The presence or absence of a given connection is represented by a TRUE or FALSE value in a boolean field. For example, a database entity having a TRUE value in its "potable water" field represents an interface plate incorporating a potable water connection. The following set of CAPS statements enforces the constraint that no interface plate may substitute for another unless the substituting plate has at least those interfaces which are present on the plate for which it substitutes.

```
allow all
for avionics_air disallow not avionics_air
for nominal_power disallow not nominal_power
for high_power disallow not high_power
for fire_detection disallow not fire_detection
for data_management disallow not data_management
for thermal_control disallow not thermal_control
for hygiene_water disallow not hygiene_water
for nitrogen disallow not nitrogen
for potable_water disallow not potable_water
for hygiene_waste disallow not hygiene_waste
```

The first statement, "`allow all`", initially permits all substitutions. The subsequent statements restrict the substitutability relation until it satisfies the requirement. A cost analysis undertaken at this point will consider the elimination of any interface plate design which incorporates a set of interfaces which is a subset of those offered by some other design. Whether or not CAPS actually recommends the elimination of such a design depends on the cost information conveyed to it.

4 Communicating Cost Information to CAPS

CAPS has two separate mechanisms for determining costs. One of those is the default costing mechanism, derived from the cost functions used in NASA's System Commonality Analysis Tool (SCAT) (see [2] and [5]). It consists of a set of relatively fast compiled-in cost functions. These functions are fairly standard and will not be discussed here. It is the data-dependent nature of costs that is of interest in the context of knowledge representation, and in recognition of that

fact CAPS incorporates a facility for describing highly tailored domain-specific cost information.

The feature of CAPS which facilitates the tailoring of cost information is the same defined variable feature discussed above, along with a set of predefined functions which have meaning only in the context of a cost analysis. A CAPS cost analysis has the task of grouping the elements of a database into subsets which are *components* of a partition. A partition of the database into such subsets amounts to a proposed solution to a commonality analysis problem. Each such subset has a distinguished representative which is proposed as a substitute for each of the other objects represented in the subset. During a typical CAPS cost analysis, many such subsets and representatives will be considered. The cost function employed by CAPS, whether it is a default cost function or user-defined, allows CAPS to attach a cost to each (subset, representative) pair. Since there is a need to identify costs associated with an entire component, rather than a single entity within the database, special-purpose functions must be employed. Following are a few such functions:

cmax(<*field name*>)	Maximum over all values of the given field within the component.
cmin(<*field name*>)	Minimum over all values of the given field within the component.
csum(<*field name*>)	Sum of all values of the given field within the component.
csize	Number of objects in this component.

Consider, for instance, the following user-defined cost function:

```
define linear_cost as ddt&e + csum(quantity)*unitcost
```

In this example, (1) "ddt&e" represents the design, development, test and engineering costs associated with the object, (2) "quantity" represents the number of copies of the object which must be produced, and (3) "unitcost" represents the marginal cost of producing each item, assuming no learning curve is used.[1] All three are field names. To make CAPS switch from its default cost function to this user-defined function, we use the following statement:

```
use linear_cost
```

[1] CAPS' default cost functions are capable of incorporating a learning curve, and facilities are provided for building a learning curve component into a user-defined cost function as well.

During the analysis of the cost of a given component, the fields "ddt&e" and "unitcost" will apply only to the chosen representative, whereas "quantity" will range over the entire component, since it is used as an argument to "csum".

As a final example, consider the interface plates database discussed in the foregoing section. In that example, the substitutability relation was designed so that plate A was allowed to substitute for plate B if B's set of interfaces was a subset of the set of interfaces on plate A. This assumption precludes the possibility of altering plate A in such a way that it incorporates additional interfaces in order to take on the functionality of plate B, or of manufacturing a new interface plate incoroporating all interfaces present on both plates. If that approach is taken, it is no longer necessary to restrict substitutability so strictly. We simply adjust our concept of what it means for one plate to "substitute" for another. Regardless of which plate is chosen as the "representative" of a given component, it is assumed that a plate will be produced which incorporates the entire set of interfaces present on all plates in that component. We need, of course, a way of measuring the cost of producing such a plate. The following sequence of CAPS statements defines a cost function which is compatible with this strategy.

```
define combined_interfaces as
    cmax(avionics_air)     + cmax(nominal_power) +
    cmax(high_power)       + cmax(fire_detection) +
    cmax(data_management) + cmax(thermal_control) +
    cmax(hygiene_water)    + cmax(nitrogen) +
    cmax(potable_water)    + cmax(hygiene_waste)
define new_plate_cost as ddt&e +
    csum(quantity) * combined_interfaces
use new_plate_cost
```

What makes the formula work is the fact that boolean attributes are assumed to have numerical values: 0 for false and 1 for true. Thus "cmax(avionics_air)" has the value 1 if any of the interface plates in the component under consideration has an avionics air cooling interface. The parenthesized sum therefore amounts to a count of interfaces which would need to be present in any plate used as a substitute for all those present in the current component. Weighting factors can easily be added to the various terms to provide a more realistic estimate of the cost.

We can restrict the total number of interfaces on a given plate to a fixed number, say 7, using a "choke term" which causes the cost function's value to become unacceptably large should the program attempt to combine a set of plates which would require as its substitute a plate having more than that number of interfaces. Such a choke term appears in the following example, where the cost function is

identical to the above except for the addition of the choke term.

```
define choked_plate_cost as ddt&e +
    csum(quantity) * combined_interfaces +
    1.0E9*(combined_interfaces>7)
use choked_plate_cost
```

In this scenario, we no longer need to restrict substitutability in any way. A simple "allow all", then, would suffice to communicate to CAPS the lack of any constraints. That would not be the most prudent way to enter the cost analysis, however, because fewer constraints typically mean a more lengthy analysis. A natural way to constrain substitutability is to always choose the plate with the larger number of interfaces. This can be done as follows.

```
define number_of_interfaces as avionics_air + nominal_power +
             high_power + fire_detection + data_management +
             thermal_control + hygiene_water + nitrogen +
             potable_water + hygiene_waste
for number_of_interfaces=x allow number_of_interfaces>=x
```

After the above constraint is imposed, the solution proposed by CAPS will always use as a representative for each component the interface plate needing the fewest additional interfaces in order to serve as a replacement for all other plates in that component.

5 Conclusion

CAPS incorporates into its design notations and operations uniquely suited for describing commonality constraints and cost information in preparation for a comparative cost analysis. The notations are intuitive and easily understood, yet powerful enough to make CAPS applicable to a broad range of commonality analysis problems. For more complete information on the CAPS tool, see [4] or contact the author for more recent documentation. Generalizations of the CAPS notation to more general database applications are presented in [6].

References

1. Clocksin, W. F. and Mellish, C. S. "Programming in Prolog". New York, Springer-Verlag, 1981.

2. Marshall Space Flight Center. Commonality Analysis Study, User Manual for the System Commonality Analysis Tool (SCAT), D483-10064, March 1987.

3. Thomas, L. D. "A Methodology for Commonality Analysis, with Applications to Selected Space Station Systems". Dissertation, The University of Alabama in Huntsville, Huntsville, AL, 1988.

4. Yeager, D. P. "Development of a Prototype Commonality Analysis Tool for use in Space Programs". Section XXXI of NASA Contractor Report CR-183553, Research Reports - 1988 NASA/ASEE Summer Faculty Fellowship Program. George C. Marshall Space Flight Center, 1988.

5. Yeager, D. P. "A Formalization of the Commonality Analysis Problem and Some Partial Solutions", BER Report Number 454-69, The University of Alabama, February 1989.

6. Yeager, D. P. "A History-Sensitive Approach to the Description of Binary Relations", Structured Programming, vol. 10, no. 3, pp. 157-163.