MASSACHUSETTS INSTITUTE OF TECHNOLOGY

ARTIFICIAL INTELLIGENCE LABORATORY

Working Paper 250 September, 1983

Virtual Inclusion

David Chapman
Philip E. Agre

Abstract

Several recent knowlege-representation schemes have used *virtual copies* for storage efficiency. Virtual copies are confusing. In the course of trying to understand, implement, and use Jon Doyle's SDL virtual copy mechanism, we encountered difficulties that led us to define an extension of virtual copies we call *virtual inclusion*. Virtual inclusion has interesting similarities to the environment structures maintained by a program in a block-structured language. It eliminates the clumsy *typed part* mechanism of SDL and handles properly a proposed test of sophisticated virtual copy schemes.

Introduction

If your program frequently creates slight specializations of representations, you'd like to avoid having it always copying over the whole representation and modifying the copy; much better to build a structure that says "this one is just like that one except ...". Such a structure is called a *virtual copy* [Fahlman PhD]. Virtual copies are confusing. In the course of trying to understand, implement, and use Jon Doyle's SDL virtual copy mechanism, we encountered difficulties that led us to define an extension of virtual copies we call *virtual inclusion*. Virtual inclusion has interesting similarities to the environment structures maintained by a program in a block-structured language. It eliminates the clumsy *typed part* mechanism of SDL and handles properly a proposed test of sophisticated virtual copy schemes.

Doyle's SDL [Doyle Phd] is a "structural description language" historically derived from Weyhrauch's FOL [Weyrauch Prolegomena]. The basic unit of representation in SDL is the *theory*. Stripped of inessentials, a theory consists of a set of *statements*, or first-order predicate calculus formulae; and a set of *attachments*, which can be thought of as bindings or model-theoretic interpretations for symbols mentioned in the formulae. One theory can contain another as a virtual copy, meaning that the first theory appears to include all the statements of the included theory. In implementation, the statements are stored only once, in the copied theory, and the copying theory contains only a virtual-copy pointer to the other. Similarly, symbols in the copying theory will appear to have the same attachments that they do in the copied theory, although the attachment is really only made once. (In case the copying theory already has an attachment for a symbol that has an attachment in the copied theory, Doyle has its apparent value in the copying theory be the local one.)

Reflexive Closure and Virtual Inclusion

The example that originally motivated us is that of representing the idea of the reflexive closure of a relation. Recall that given a relation r, rc is its reflexive closure if rc holds between two objects just when r does or when the two objects are identical. That is,

```
(forall (x y)
  (equivalent (or (= x y) (r x y))
              (rc x y)))
```

If we often make reflexive closures of relations, we will end up with many slightly different versions of the statement above, with different symbols in place of r and rc. This sort of similarity is what virtual copies are supposed to help with: we'd like to have a theory reflexive-closure contain one copy of the statement, and make each specific reflexive closure a virtual copy of the reflexive-closure theory. The difficulty is that each reflexive closure mentions different relations; and the abstract reflexive-closure theory must mention some specific ones. Virtual copies as usually understood can't help.

Our solution should be obvious. We extend virtual copy to allow renamings of symbols. Then each reflexive closure theory can rename the symbols used in the abstract reflexive-closure theory to whatever specific relations it concerns. We call virtual copy with renamings *virtual inclusion*. (The reason for this name will become clear in the section on typed parts.)

Virtual inclusion has been implemented and works. Here is the actual SDL-with-virtual-inclusion code for the reflexive closure example. First we define the abstract theory of reflexive closures.

```
(deftheory reflexive-closure)
```

This statement is true in the reflexive-closure theory:

```
(statement reflexive-closure
           (forall (x y)
                   (equivalent (or (= x y)
                                   (closed-over-relation x y))
                               (relation x y))))
```

Now the theory r-theory, over which we will close.

```
(deftheory r-theory)

(statement r-theory (r a b))

(statement r-theory (not (r b a)))
```

Now we form the closure.

```
(deftheory rc-theory)
```

We virtually include reflexive-closure in rc-theory with the appropriate renamings: relation (the closure) is renamed to rc, and the closed-over-relation is r.

```
(include! (get-theory 'reflexive-closure)
          (get-theory 'rc-theory)
          '((relation rc) (closed-over-relation r)))
```

Now effectively rc-theory contains the statement

```
(statement rc-theory
           (forall (x y)
                   (equivalent (or (= x y) (r x y))
                               (rc x y))))
```

We include r-theory in rc-theory so that rc-theory will know when r holds.

```
(include! (get-theory 'rc-theory)
          (get-theory 'r-theory)
          '())
```

We can test the representation using a simple theorem prover, asking it for things of the form (rc ? ?):

```
(find-examples '(rc ? ?) (get-theory 'rc-theory))

=> (((RC A A) (RC A B) (RC B B))
    ((RC B A)))
```

The first sublist above contains things provably true, and the second things proved false. The explict statements about r were used to prove (rc a b) and (not (rc b a)); the statement from reflexive-closure proved (rc a a) and (rc b b).

Typed Parts

Doyle uses virtual copies to implement *typed parts*. Suppose that you are describing logic circuits, and you have a theory of an adder. (See [Doyle PhD] pp 72-77.) You want a theory that describes a doubler. The doubler theory can have as a part a theory of an adder (a theory that is a virtual copy of an adder). The doubler can further specify that the two inputs of the adder are to come from the same source (thus effectively making the adder into a doubler).

How is the doubler theory to talk about the adder that is a part of it? Doyle's solution is to use attachment. He attaches the theory of the adder to a symbol in the doubler theory. Then he introduces a special piece of syntax that extends the predicate calculus to allow reference to attachments. These he calls pathnames. For example, (the gribble praxoon) refers to the object that is attached to the symbol gribble in the theory that is attached to the symbol praxoon in the current theory. Once all this mechanism is in place, he can attach the adder theory (a virtual copy of the prototype adder theory) to the symbol adder in the doubler theory. And then he can assert

```
(statement doubler
           (= (the first-input adder) (the second-input adder)))
```

so that if either input is given a value, reasoning procedures will find the same value for the other, and produce twice that value as the output. Further asserting

```
(statement doubler (= input (the first-input adder)))

(statement doubler (= output (the output adder)))
```

will make it so that the doubler's input and output are logically connected to those of the adder; asserting a value for the doubler's input will now cause a value to be found for its output (if the prototype adder theory is correct, and if we have a deduction engine for equality).

The perceptive reader will note that we have gone to considerably more work than necessary to describe the doubler. There is no reason that we could not have simply made doubler a virtual copy of the prototype adder theory and added the extra constraint that the two inputs be equal. There is a common situation in which all this complex mechanism is required, though: when we want two typed parts of the same type. Doyle's example is a quadrupler made out of two doublers in series. The two doublers must be distinct, because one will produce twice the input to the quadrupler, and the other should produce four times that quantity. Evidently, making the quadrupler a virtual copy of the doubler twice is not going to work. Instead, Doyle makes two virtual copies of the doubler (the typed parts) and attaches them to distinct symbols in the quadrupler theory.

There are at least three difficulties with typed parts. First is that they are ugly that if you only have one part of a given type, simple virtual copy will suffice, but if you have more than one, a complicated kludge (involving extensions to predicate calculus) is required. It would be nicer if you could somehow make the theory with parts a virtual copy several times. Second, this scheme requires a fair amount of memory to store the virtual parts, which don't really do anything — they are just placeholder symbols, and all you care about them is that they are distinct. Third, it seems that there is a confusion of semantic levels: to reason about a part, such as a doubler, we attach the theory of the doubler to a symbol. But what if you want to reason not about the part but about the theory of the part? SDL leaves you in the lurch. Since SDL is in fact supposed to support this sort of reflexive reasoning, this might be quite serious.

Typed parts can be wholly eliminated using virtual inclusion. It is not meaningful to make something a virtual copy of something else several times; but it is meaningful to virtually include a theory in another several times, if there are different renamings.

Here is Doyle's quadrupler example, using virtual inclusion.

```
(deftheory adder)

(statement adder (= (+ a b) s))
```

The doubler will virtually include the adder, *but with* a *and* b *renamed to the same quantity* x*!* Notice that we don't have to make any assertions about equality of parts. This drastically decreases the amount of reasoning necessary to quadruple a quantity; the propagation of values back and forth between the part theories and the quadrupler theory is made unnecessary.

```
(deftheory doubler)

(include! (get-theory 'adder)
          (get-theory 'doubler)
          '((a x) (b x) (s 2x)))
```

The quadrupler has two inclusions of the doubler with different renamings.

```
(deftheory quadrupler)

(include! (get-theory 'doubler)
          (get-theory 'quadrupler)
          '((x in) (2x temp)))

(include! (get-theory 'doubler)
          (get-theory 'quadrupler)
          '((x temp) (2x out)))
```

Now we can test this by retrieving all the statements virtually included in quadrupler:

```
(theory-statements (get-theory 'quadrupler))
=> ((= (+ IN IN) TEMP) (= (+ TEMP TEMP) OUT) ())
```

These two statements (one derived from each copy of the doubler) do in fact guarantee that out is four times in.

Quantification and Attachment

Bawden, Hillis, and McAllester, in unpublished work, pose as tests of virtual copy schemes the proper compression of the following three quantified statements:

o Everyone has a nose (and nobody has anyone else's nose).

o Everyone has a mother (who may or may not be the same as anyone else's mother).

o Everyone hates the phone company (and everyone hates the very same phone company). (The imminent antitrust fragmentation of AT&T will soon make this claim untrue!)

Many obvious ways of compressing these collections of statements change the semantics of one or more of them. More subtle schemes may fail if the identities of everyone's nose or mother is not known. The "mother" example is particularly difficult.

These problems can be solved in SDL using unattached symbols. If person has an unbound mother pointer then when you go to instantiate it as Sue, she too gets an unbound mother pointer. Later on you might be able to bind the pointer when you find out that her mother is Mary.

The power of virtual inclusion is not really necessary for this problem; virtual copy could do the job, thought it works out a little less nicely. We give the SDL-with-virtual-inclusion solution as the final example in this paper.

```
(deftheory person)

(statement person (hates person telco))

(statement person (has-nose person nose))

(statement person (has-mother person mother))
```

Everyone hates the same telephone company. attach! is a function of three arguments that makes the interpretation of the first argument (a symbol) be the second argument in the theory that is the third argument.

```
(attach! 'telco 'ma-bell (get-theory 'person))
```

But Caesar has his own nose.

```
(deftheory caesar)

(attach! 'caesars-nose 'nose-1 caesar)

(include! (get-theory 'person)
          (get-theory 'caesar)
          '((person caesar) (nose caesars-nose)))
```

Tests:

```
(try-to-show '(has-nose caesar caesars-nose) (get-theory 'caesar))
=> :TRUE

(attachment 'caesars-nose (get-theory 'caesar))
=> NOSE-1

(try-to-show '(hates caesar telco) (get-theory 'caesar))
=> :TRUE

(attachment 'telco (get-theory 'caesar))
=> MA-BELL
```

Caesar does indeed have a mother:

```
(find-examples '(has-mother caesar ?) (get-theory 'caesar))
=> (((HAS-MOTHER CAESAR MOTHER)) ())
```

But we don't know who she is.

```
(attachment 'mother (get-theory 'caesar))
=> :UNKNOWN
```

Discussion

An interesting feature of virtual inclusion is its similarity to the environment structures built up by an interpreter for a block-structured language. (Scheme environment diagrams in fact probably influenced the discovery.) Brian Smith has argued in unpublished papers for the construction of a unified declarative/procedural language that would both represent and compute. In arguing for the plausibility of this endeavor, he cites the many similar mechanisms (such as variables, intension, and designation) that are used in both declarative and procedural languages. Perhaps enviroment structures should be added to this list.

The virtual inclusion scheme for implementing virtual copies does not allow one to modify the copied theory in as general a way as one might like. We do not know how to implement copy modification in any general way. Doyle describes a scheme which assumes that a proposition's membership in a theory is determined by whether an proposition to the effect that that proposition is a member of the theory is believed by the system, but we have not tried to implement it.

SDL-with-virtual-inclusion was implemented by Chapman as a representation language for a learning program described in [Chapman Naive]. It was soon abandoned, because of a fundamental problem with SDL (rather than with virtual inclusion). The problem is mentioned in passing by Doyle: SDL is fine at statically representing things, but as described it provides no way of getting information out.

> "The fiction about the representational scheme presented here is actually a symptom of a larger incompleteness in this thesis, namely the lack of database retrieval procedures altogether. McDermott, Fahlman, and others have argued for a separation between database retrieval and problem solving, where database retrieval consists of applying automatic, quick procedures which adequately handle almost all queries (the routine cases), and problem solving consists of applying carefully controlled inference procedures to ferret out the desired information that routine procedures miss..." ([Doyle Phd] p. 63).

Thus, SDL takes a view of the world in which on the one hand there is a puzzle in which facts have been hidden, and on the other hand is a general purpose procedure for "ferreting" out the encoded information. Given that SDL is basically a modularization of first order predicate calculus, that general purpose procedure must be essentially a theorem prover. Essentially nothing is known about general theorem proving. The try-to-show and find-examples procedures used above in fact call a resolution theorem prover, which is known to be a bad idea when one's problem can't be fit in a thimble. Lacking any better theory of how to get knowlege out, SDL was eliminated entirely. (It is worth noting that Doyle never made any strong claims for SDL; it was incidental to the main thrust of his thesis, which continues to inspire us.)

What *are* virtual copies anyway? Bawden, Hillis, and McAllester sensibly suggest that attempting to give virtual copies any semantic status in one's representation is so difficult that one is best off keeping them at the implementation level, where they can do a fine job of data compression. (Virtual inclusion might compress the quantified examples above, but they certainly don't represent them in any robust way.) One might argue that virtual copies are in fact redundant: any regularity in one's database that might be compressed to any great advantage with virtual copies ought to be understood and represented explicitly with semantically well-understood representation constructs like quantification and A-Kind-Of (sic). Heavy use of virtual copies in place of more explicit constructs for representing part/whole relationships produces large "flat" theories. While this might be tolerable when there are three virtual inclusions of the theory of gravity in a theory about interactions between Jupiter, Saturn, and the sun, it would be a computational disaster when applied recursively in a representation of, say, a VLSI circuit. Whether virtual copies in general or virtual inclusion in particular have any role after these lessons are learned is an empirical matter.

References

[Chapman Naive] David Chapman, *Naive problem-solving and naive mathematics*. MIT AI Lab Working Paper 249, June 1983.

[Doyle PhD] Jon Doyle, *A Model for Deliberation, Action, and Introspection*. MIT AI TR 581, May 1980.

[Fahlman PhD] Scott E. Fahlman, *NETL: A system for representing and using real-world knowledge*. Cambridge: MIT Press, 1979.

[Weyhrauch Prolegomena] Richard W. Weyhrauch, *Prolegomena to a Theory of Mechanized Formal Reasoning*. Stanford AI Memo 315.