This paper develops the notion of temporal abstraction, used originally for the automatic understanding of looping constructs, to account for a class of recursive programs involving side effects upon a relational data base. The programs may involve compositions of several side effects, and these side effects can occur either during descent or upon ascent from recursive calls.

I INTRODUCTION

The concept of 'temporal abstraction' was developed by Waters [5], and Rich & Shrobe [4] to describe the variables enumerated in loops as a set of objects which could be manipulated as a whole. We apply this principle to recursive procedures operating on threaded data structures and use it to understand compositions of several side effects. Our analysis is part of a larger project designed to help novice programmers understand their buggy programs. Students learning a LOGO-like language called SOLO [1], in which a procedure can only side effect a global data base which is a directed graph. Here is a sample problem, of the kind posed to our students: Define a procedure INFECT which would convert a data base such as the one shown in Fig. 1 to the one shown in Fig. 2 if invoked as INFECT ANDY.

```
|-------| INOCULATED <---- | is kisses kisses
|       |                   |
| -----> | BARBARA<-------| COLIN <-----| DIANA
  kisses kisses
```

Fig. 1

```
|-------| INOCULATED <---- | is gets
|       |                   |
| -----> | BARBARA<-------| COLIN <-----| DIANA
  gets
```

Fig. 2

II A CLASS OF RECURSION SCHEMATA

In a recursive procedure such as INFECT, a side effect (conditional or unconditional) may occur logically at any of five locations, depending on the juxtaposition of the side effect, the recursive call, and the termination test. Here is a skeleton of the INFECT procedure, with the five possible locations of side effect occurrences shown underlined (not all five can coexist, and the SOLO user must make careful use of the control-flow keywords CONTINUE and EXIT to obtain certain combinations):

```
TO INFECT /X/
  1 EXAMINE /X/
  2 CHECK /X/ KISSES ?Y
    2A If present: INFECT "Y; EXIT
    2B If absent: EXIT
  TO EXAMINE /X/
    1 CHECK /X/ IS INOCULATED
      1A If present: EXIT
      1B If absent: NOTE /X/ GETS FLU; EXIT

INFECT recursively generates successive nodes along the thread of 'kisses' relations (steps 2 and 2A), and invokes EXAMINE at each node. EXAMINE conditionally side effects each node, i.e. asserting that /X/ GETS FLU only when the triple /X/ IS INOCULATED is absent.

This problem is typical of a large class of tasks given to beginning SOLO users. Students may adopt a variety of methods for tackling the stated problem, combining side effects to achieve the desired result. The next two sections describe how we cope with different varieties of recursion and how the accumulation of side effects is represented. Section IV then illustrates how deviant cases are handled.
TO INFECT /X/
...
<initial>
...
CHECK /X/ KISSES ?Y
If present: <pre-rec>; INFECT *Y; <post-rec>
If absent: <termination>
...
<final>

These occurrences are depicted schematically in Fig. 3, which shows a recursive procedure P partitioned into its possible constituents. The effect of each solid box in the figure, in accordance with the notion of temporal abstraction, is to add or delete some set of triples. The overall effect of P will be the composition of these, as described in section III. 'Rec' is short for 'recursion', 'self' is the actual recursive invocation of P, and 'get-next' is an implicit enumeration function which retrieves the next node in the thread (this happens automatically during pattern-matching in SOLO, e.g. when Y is bound to BARBARA, then COLIN, then DIANA).

Fig. 3: The recursion schema for thread enumeration

Any of the effect steps may be conditional, composite, or missing altogether. The restrictions for recognising a program as an instance of this schema are: (1) P has a parameter x which is the beginning of a thread; recursive invocations of P thus enumerate successive nodes along the thread; (2) the recursion and termination steps involve the enumerated node and relation R, where R is non-cyclic and one-to-one or many-to-one in the database (we don’t deal with one-to-many mappings of R, which would exist if, say, the triples ANDY KISSES BARBARA and ANDY KISSES DIANA were both present in the database); (3) the side effects do not alter the thread and the side effected node can only be reached from the enumerated node via one-to-one relations.

The important insight of temporal abstraction is that the nodes enumerated during recursion can be dealt with as a set. Thus, any of these steps has an effect which can be represented as a set of triples (which we call 'db-set'). The termination step is the degenerate case of a singleton set. We describe db-sets as follows:

\[
[\text{db-set} \quad \text{typical member:} \quad (\text{filter} \Rightarrow \text{side effect})
\]
\[
\text{init:} \quad (<\text{enumerated node}> . <\text{first value}>)
\]
\[
\text{rec rel:} \quad \text{thread link}
\]
\[
\text{termination:} \quad (\text{ABSENT} \quad ([<\text{ref}] \quad \text{thread link} \quad ?)]
\]

where
\[
\text{filter} := T | \text{<simple filter>} | (\text{OR} \quad \text{<conjunctive filter>} \quad \ldots)
\]
\[
\text{<simple filter>} := \text{(PRESENT} \quad \text{<triple}> \quad \text{)} | (\text{ABSENT} \quad \text{<triple>})
\]
\[
\text{<conjunctive filter>} := \text{(AND} \quad \text{<simple filter>} \quad \ldots)
\]
\[
\text{<side effect>} := (+ \quad \text{<enumerated node}> \quad \text{<link}> \quad \text{<node>}) | (- \quad \text{<enumerated node}> \quad \text{<link}> \quad \text{<node>})
\]
\[
\text{<ref>} := \text{<enumerated node> (get <ref> <link>)}
\]

By (get <node> <link>) we denote the reference from <node> via <link>. Thus <ref> is the n-fold composition of the reference from the <enumerated node> (called e) along <link>. For example, the node BARBARA in Fig. 1 can be referenced by (get ANDY KISSES), whereas the node DIANA can be referenced by (get (get ANDY KISSES) KISSES). An instantiated schema does not, of course, refer to specific nodes in the data base, but rather to a generalised description of a typical node along the thread.

How, then, are db-sets derived from an instantiated recursion schema? We fill the slot 'typical member' from the effect of the step (e.g. a NOTE of some triple is a + with its first argument replaced by a <ref> involving e). If the effect is unconditional, the filter is T, otherwise the condition is taken as the filter. The 'init' slot is the first value that e will take. Enumeration stops when termination is true. Steps
which work on the entire thread have
\((\text{ABSENT } [e <\text{thread link}> ?])\)
as their termination condition, whereas those
working on the butlast of the thread have
\((\text{ABSENT } [(\text{get } e <\text{thread link}> ) <\text{thread link}> ?])\)
as their termination condition.

Consider the instantiated schema for
\text{SOLUTION-1} above, which contains only an initial
(conditional) effect. Its effect description is as follows:

\[
\begin{align*}
\text{[db-set]} \\
\text{typical member: } (\text{ABSENT } [e \text{ IS INOCULATED}]) \Rightarrow (e \text{ GETS FLU}) \\
\text{init: } (e . x) \\
\text{rec rel: KISSES} \\
\text{termination: } (\text{ABSENT } [e \text{ KISSES ?}])
\end{align*}
\]

III  COMPOSITION OF RECURSIVE SETS
BY SYMBOLIC EVALUATION

In general, a recursive procedure may comprise
some combination of effect steps. To describe the
net effect of the entire procedure, the individual
effects must be composed. For instance, the
addition of a set of triples followed by the
conditional deletion of some elements should have
the net effect of asserting set elements of
the set conditionally upon the negation of the
condition for deletion (e.g. see \text{SOLUTION-2}
below). This simplification is done during
symbolic evaluation.

Although the db-set for each step is a
temporal abstraction (and hence ignores the order
in which the nodes are enumerated), compositions of
side-effects are sensitive to temporal order.
Thus, we first compose those effects which occur on
the descent and then those which occur on the
ascent. The effects are still dealt with as
db-sets, i.e. the inner details of the enumeration
sequence are ignored. All we need to worry about
is whether the db-set is of the 'descending' or
'ascending' variety, which we know from its position
in the instantiated schema.

Composition proceeds as follows: At a node in
the symbolic evaluation tree (called \text{S-node}) where
a db-set is to be asserted, we grow a branch with a
description of the range of values that could be
taken by the enumerated node. The range \(\{v | v \in R^*(x)\}\) says that \(e\) can take the value of all the
nodes in the transitive closure of \(R\) starting at \(x\).
At the \text{S-node} at the end of this branch, the
typical member of the db-set is asserted. If there
is a (non-T) filter, we split into two branches,
one with the condition and the other with its
negation, and add the side effect to the \text{S-node} at
the end of the branch where the condition is true.

The next db-set is dealt with in the same way
at all terminal \text{S-nodes} grown so far. If it has a
filter, we test it at each \text{S-node}: if the sets
have the same range, it is possible to show
whether the condition must always hold or that
it can never hold, thus saving the split. If the
ranges overlap, we introduce one branch for the
overlapping range and others for the
non-overlapping parts. On the overlapping sets it is
then possible to test conditions or apply rules
for cancellation (i.e. a \text{NOTE} followed by a \text{FORGET}
of the same triple has no net effect) and
overwriting (i.e. a \text{NOTE} following another \text{NOTE}
with the same triple has no further effect).

A special case arises if two consecutive
assertions of sets are interleaving (i.e. they
have the same time of execution of effect), and
the second set’s enumerated node (\(e_2\)) ‘runs ahead of’
the first set’s enumerated node (\(e_1\)). We say that
\(e_2\) ‘runs ahead of’ \(e_1\) if \(e_2\) is an \(n\)-fold
composition of the reference from \(e_1\) along the
thread-link. Since the effect on \(e_2\) occurs before
the effect on \(e_1\), we reverse the order of
composition of both sets and apply the
simplifications described above. Finally, starting
from the terminal \text{S-nodes} we collect effects and
conditions for all nodes with the same net-effect
into one db-set with the appropriate filter. The
result is the description of the program used for
comparison with the ideal effect description.

IV  EXAMPLES

Here are two alternative solutions to the
problem posed in section 1. One has a bug.

\text{SOLUTION-2 (Successive sweeps)}:

\begin{itemize}
  \item \text{TO INFECT } /X/ \\
  \hspace{1em} 1 \text{ CONTAMINATE } /X/ \\
  \hspace{1em} 2 \text{ DECONTAMINATE } /X/ \\
  \item \text{TO CONTAMINATE } /X/ \\
  \hspace{1em} 1 \text{ NOTE } /X/ \text{ GETS FLU} \\
  \hspace{1em} 2A \text{ If present: CONTAMINATE } *Y; \text{ EXIT} \\
  \hspace{1em} 2B \text{ If absent: EXIT} \\
  \item \text{TO DECONTAMINATE } /X/ \\
  \hspace{1em} 1 \text{ CHECK } /X/ \text{ IS INOCULATED} \\
  \hspace{1em} 1A \text{ If present: FORGET } /X/ \text{ GETS FLU}; \text{ CONTINUE} \\
  \hspace{1em} 1B \text{ If absent: CONTINUE} \\
  \hspace{1em} 2 \text{ CHECK } /X/ \text{ KISSES } ?Y \\
  \hspace{1em} 2A \text{ If present: DECONTAMINATE } *Y; \text{ EXIT} \\
  \hspace{1em} 2B \text{ If absent: EXIT} \\
\end{itemize}

\text{SOLUTION-3 (Ascending conditional side effect)}:

\begin{itemize}
  \item \text{TO INFECT } /X/ \\
  \hspace{1em} 1 \text{ CHECK } /X/ \text{ KISSES } ?Y \\
  \hspace{1em} 1A \text{ If present: INFECT } *Y; \text{ CONTINUE} \\
  \hspace{1em} 1B \text{ If absent: EXIT} \\
  \hspace{1em} 2 \text{ CHECK } /X/ \text{ IS INOCULATED} \\
  \hspace{1em} 2A \text{ If present: EXIT} \\
  \hspace{1em} 2B \text{ If absent: NOTE } /X/ \text{ GETS FLU}; \text{ EXIT} \\
\end{itemize}

In \text{SOLUTION-2}, the plan diagram for \text{INFECT}
matches a schema for a conjoined effect. The plan
diagram for the first of these, \text{CONTAMINATE},
matches the recursion schema for thread
composition: an unconditional side effect
occurring only in the 'initial effect' slot of the
schema. It thus has the net effect:
The second of the conjoined effects, DECONTAMINATE, matches the same schema, except that the side effect in the 'initial effect' slot is conditional. This yields the following net effect description:

\[
\text{for all } \{e \mid e \in \text{KISSES}(x)\} \\
(+ [e \text{ GETS FLU}])
\]

During symbolic evaluation, the two S-nodes are recognized as having the same range, so the evaluator simplifies their combined effect to be:

\[
\text{for all } \{e \mid e \in \text{KISSES}(x)\} \\
(- [e \text{ GETS FLU}])
\]

which is precisely the intended effect of the ideal INFECT procedure.

SOLUTION-3 has a plan diagram which matches that of the recursion schema with just a (conditional) 'post-rec' effect. The reader may wish to verify that this conforms with the skeleton depicted at the beginning of section II and the schema shown in Fig. 3 (the CONTINUE at step 1A in effect places step 2 in the 'post-rec' position). Notice that in this solution the first thing that normally happens is the recursive invocation of INFECT (step 1A), which means that the side effect only happens when ascending. Our schema knows that a 'post-rec' effect on its own fails to reach the final node in the thread (in this case it is due to the EXIT at step 1B). This is instantiated from the schema's canned effect description as follows:

\[
\text{for all } \{e \mid (\text{get } e \text{ KISSES}) \in \text{KISSES}(x)\} \\
(- [e \text{ IS INOCULATED}]) \Rightarrow (+ [e \text{ GETS FLU}])
\]

That is, the conditional side effect is perpetrated only on the butlast of the thread running from x via KISSES. For the example of Fig. 1, SOLUTION-3 happens to work. However, a counter-example can be generated for the student in the following way: generate a thread in which the final node (f) of the thread satisfies the condition specified in the schema's effect description, i.e. \((\text{ABSENT } [f \text{ IS INOCULATED}])\). For Fig. 1, this would amount to deleting the IS link between DIANA and INOCULATED. In such a case, SOLUTION-3 will fail. This counter-example can be used to point out the inherent flaw in the student's solution.

### V CONCLUSION

Temporal abstraction provides us with a powerful mechanism for reasoning about side effects on sets of data objects. Our method of composing several such sets lets us analyse recursive procedures involving side effects on threaded data structures. Static set descriptions, which are the essence of temporal abstraction, can be combined using simplification rules derived from symbolic evaluation techniques. A library of recursive schemata enables us to analyse a range of students' programs, to combine set descriptions in a sensible way (e.g. composing descending and ascending effects in the right order), and to generate tailor-made counter-examples for programs which might 'work' by accident.

### REFERENCES


