J-graphs and J-code

We divide the problem of generating verification conditions into two parts. We first translate the program to be verified into a representation we call a J-graph, and then trace out the paths in the J-graph. We have defined a textual language we call J-code which is used to represent J-graphs. The decompiling pass of the verifier generates J-code from Icode. The path tracer then parses J-code and builds a J-graph, then traverses the J-graph to produce verification conditions.

The J-graph is best thought of as a nondeterministic loop-free program. J-graphs contain the primitives **NEW, SPLIT, WHEN, BRANCH, JOIN, HANG, and REQUIRE.**

The NEW instruction is a nondeterministic, simultaneous assignment statement. The format of a NEW instruction is

```
NEW     <variable-list> <formula>         .
```

When a NEW instruction is executed, the variables in the *<variable-list>* **are upated in such a way as to make the** *<formula>* **true. If** *v* **is a formula in the variable list, the notation** *v.old* **may be used to denote the value of the variable before the** *NEW* **instruction is executed.**

Assignment statements and procedure calls are translated to *NEW* **instructions in a straightforward fashion. For example, the assignment statement:**

```
X := X + 1
```

is translated to:

```
NEW (X) (X = X.OLD+1)
```

To translate a procedure call, the decompiler determines every variable which could be modified as a result of the call, either by direct modification or through side effects. This list of variable forms the <variable-list> of the NEW instruction. The <formula> is derived from the EXIT condition declared with the procedure.

The SPLIT, WHEN, BRANCH, and JOIN instructions are used to model flow of control in J-graphs. SPLIT and WHEN are used when the flow of control forks off in different directions, and BRANCH and JOIN are used when it comes back together again. This notation merely allows us to represent a loop-free flowchart in a linear fashion. As an example,

```
IF X > Y THEN M := X ELSE M := M + 1;
```

would be represented by

```
SPLIT   1
WHEN X > Y    1
NEW (M) (M = X)
BRANCH     2
WHEN NOT (X > Y)    1
NEW (M) (M = M.old + 1)
BRANCH 2
JOIN 2
```

All similarly numbered statements are considered to be connected.

The REQUIRE statement is used to put verification goals into J-graphs. The format of a REQUIRE instruction is

```
REQUIRE <formula>     .
```

Whenever we must prove that some formula is always true when control reaches some point in a program, we put a REQUIRE instruction containing that formula at the corresponding point in the J-graph. For each REQUIRE statement, the path tracer will trace out all paths from the REQUIRE statement back to the beginning of the program, and for each path, it will generate and submit to the

theorem prover a verification condition.

Finally, the instruction terminates a path. An execution path which reaches a HANG instruction is not continued past that instruction.

J-graphs must be loop-free. Every execution path which starts at the beginning of the J-graph must be traced forward to a HANG instruction without reaching any instruction in the graph more than once. Since we require users to write loop invariants for all loops, we are able to break the loop at the invariant and thus end up with a loop-free graph. For the program fragment

```
REPEAT
  s1;
  INVARIANT p;
  s2;
  UNTIL b;
```

where *s1* and *s2* are blocks of code, *p* is the loop invariant, and *b* is a boolean expression, the corresponding J-code is

```
SPLIT    1
WHEN TRUE      1
BRANCH         2

WHEN TRUE      1
NEW (<list-of-variables-changed-in-loop>) p
<J-code for s2>
SPLIT    3
WHEN NOT b     3
BRANCH         2

JOIN     2
<J-code for s1>
REQUIRE p
SPLIT    4
JOIN     4
HANG

WHEN b         3
```

(The graph has been simplified for clarity by the omission of the statements usually present for proof of loop termination.) The loop invariant is proved by induction on the number of times the loop body is executed. For the base case of the induction, we must show that *p* is satisfied when the **INVARIANT** statement is reached on the first execution of the loop body. This case is handled by the path which goes back from the REQUIRE through the J-code for s1, the **JOIN 2, the first BRANCH 2 and back to the beginning of the program. For the induction step, we must show that if *p* holds when the INVARIANT statement in the original program, then when the INVARIANT is reached again, P will still hold. This case is handled by the path from the REQUIRE back through the JOIN 2, the second BRANCH 2, WHEN NOT b, the J-code for s2, the NEW statement, and back through the WHEN TRUE 1 and SPLIT 1 to the beginning of the program. Finally, the path taken by the final execution of the loop body is represented by the path starting at the end of the J-code fragment and continuing back via the WHEN b 3 statement, through the loop invariant in the NEW statement, and eventually back to the beginning of the program.**

The J-graph thus generated captures the semantics of the original program. It is worth noting that our implementation generates J-code by a process very similar to that used for generating machine code in a compiler. The process of generating J-code is much more amenable to such treatment than that of directly generating verification conditions, and allows us to draw heavily on techniques from compiler technology in our verifier implementation.

Verification Condition Generation

The semantics of the language having been captured in the J-graph, the task of the verification condition generator is primarily that of tracing out all paths back from every REQUIRE statement back to the beginning of the J-graph and generating verification conditions during the process. The verification condition formally expresses the proposition ‘‘If this path is taken through the program then the formula on the **REQUIRE** instruction will be satisfied.‘‘. As soon as the path tracer generates a verification condition, it passes the formula to the theorem prover. If the theorem prover cannot simplify the formula to TRUE, the path tracer displays a diagnostic message similar to the one below.

```
Could not prove {subscerr.pf:12}
    (i + 1) - 1 <= 99    (subscript check for "tab" 1..100)
for path:
    {subscerr.pf:5} Start of "examine"
    {subscerr.pf:8} IF->THEN
    {subscerr.pf:7} IF->THEN
```

The error message identifyies the point in the program at which the **REQUIRE** was generated, the proof goal of the **REQUIRE, and the explaination as to why the REQUIRE was generated, and the path being traced. The user is not ordinarily exposed to the verification condition itself. Our experience is that the above information is usually sufficient to allow the user to correct the problem.**

The actual verification condition is constructed from formulae on the WHEN, NEW, and REQUIREinvolves renaming of variables. A unique name is invented for each of the variables in the variable lists of NEW instructions along the path. Then one of these unique names is substituted for every variable in every formula on the path. If *v* **is a variable in a formula, to find the unique name to be substituted for** *v,search back along the path for the first* **NEW** *instruction that has v in its variable list. The unique name associated with v in that instruction is used.*

Special attention is required when v occurs in both the variable list and formula of a **NEW** *instruction. Recall that v referes to the value of the variable after the* **NEW** *is executed, and v.old refers to the value of the variable before the* **NEW** *is executed. Therefore, the unique name for v is taken from the* **NEW, while the unique name for** *v.old* **is taken from the previous NEW instruction on the path mentioning** *v* **in its variable list.**

After all the variables have been renamed, the verification condition is simply

p1 and p2 and ... pn implies q

where *q* **is the formula in the REQUIRE being processed, and the terms** *p1 pn* **are the formulas on all the other instructions on this path. It is not strictly necessary to include among the** *p* **terms formulas from REQUIRE instructions passed over during backwards tracing, but we do so to stop the user from getting an avalanche of diagnostics that all result from a single error. In effect, after a REQUIRE is passed, it is assumed to be true. This is valid since the graph is loop-free. It is also very effective in reducing the number of diagnostic messages and in speeding up the proofs.**

Optimizing verification conditions

The major motivation for our approach to verification condition generation is the ability to provide good diagnostics. However, the generation of a separate verification condition for every **REQUIRE/path pair is expensive, since the number of paths is exponential in the size of the graph. Fortunately, we have discovered some techniques to reduce the amount of computation required to process a J-graph.**

The first optimzation we call ‘‘NEW balancing’’. We define a graph to be NEW balanced if for every variable *v* **and point** *P* **in the graph, every path from the beginning of the graph to P contains the same number of NEW instructions with** *v* **in their variable lists. If a graph is NEW-balanced, the path tracer does not have to perform a renaming operation each time it processes a verification condition. Instead, renaming need only be performed only once, before any verification conditions are generated.**