Pascal-F Verifier Internal Design Document

J-code to Verification Condition Translator

Scott D. Johnson

FACC / Palo Alto

1. Introduction

The input to this program is J-code that has been augmented using the routine database. The program outputs a series of verification conditions, and the error message to print if the formula cannot be proved. This design does not call for any interaction between the formula generator and the theorem prover.

This version of the program handles the following J-statements: NEW, SPLIT/WHEN, BRANCH/JOIN, REQUIRE, HANG.

1.1 Data structures

1.1.1 Strings

center;
l.
```
TYPE StringIndex = 1..StringPoolSize;
VAR StringPool: ARRAY [StringIndex] OF CHAR;
   NextString: StringIndex;
```

The string pool is a place in which strings are kept. Strings are represented by values of type StringIndex. The string represented by the string index j is the characters beginning at StringPool[j] and running sequentially through StringPool up to, but not including the next occurrence of '/)' in StringPool. A consequence of this representation is that no string can contain slash-paren as a substring.

Strings are allocated contigously in StringPool, starting at the low-numbered indices. The variable NextString contains the lowest index of StringPool that has not been allocated to a string.

1.1.2 The Label Table

center;
l.
```
TYPE LabelType = (SplitWhen, BranchJoin);

    LabelNode = RECORD
      LabelVal: Integer;       (* The label in question *)
      Class: LabelType;        (* The "type" of the label *)
      One, Many: ^Jnode;       (* Pointers to Jnodes with this label *)
      HowMany: Integer;        (* The length of the Many list *)
      BucketMate: ^LabelNode;  (* Hash bucket list *)
      END;

VAR LabelTable: ARRAY [1..LabelTableSize] OF ^LabelNode;
   MaxLabelBucket: 0..LabelTableSize;
```

The Label Table is used during the StartGraph phase to connect up Jnodes that have the same label. An open hash table is used, keyed on the numeric value of the label. Since label values are supposed to be contiguous, a simple modulus scheme is used to map labels onto hash buckets.

There are two "types" of labels: those that go on SPLIT and WHEN instructions, and those that are used with BRANCH and JOIN. The One field points to the Jnode containing the defining instance of the label

(the SplitN or JoinN node), and the Many field points to a list of Jnodes (linked through the BranchMate or WhenMate fields) that reference the label. The HowMany field contains the length of the Many list.

In many cases the LabelTable will only be partially full. The variable MaxLabelBucket is used to keep track of the highest slot used so that when the table is traversed, the entire array need not be searched.

1.1.3 The Enviornment Pool

center;
l.

```
CONST Instance = 0 .. 1295; (* 1295 = 36*36-1 *)
    EnvIndex = 1 .. NewPoolSize;
VAR   EnvPool: ARRAY[EnvPoolSize] OF Instance;
    NextEnv: EnvIndex;
    EnvLength: Integer;
```

The purpose of the NEW instruction is to advance the instance numbers of some program variables. It is assumed that the instance number of a variable can be encoded in two extra characters appended to variable names. In various parts of the J-graph, it is necessary to keep a vector of the current instance number These vectors are called "enviornments." They are represented by a pointer into NewPool. If X is an EnvIndex, the instance numbers for a variable whose index is V is retrieved by EnvPool[X+V]. All enviornments are the same length the same length (EnvLength), though that length is a function of the J-unit being processed. NextEnv is the first unused EnvIndex in the pool.

1.1.4 Variables

center;
l.

```
TYPE VariableIndex = Integer;
   Variable = RECORD
     Name: StringIndex;
     Type: StringIndex;
     Index: VariableIndex;     (* Used to consecutively number variables *)
     BucketMate: ^Variable;    (* Used by hash lookup *)
     StatusStack: ^StatusNode; (* Used during VC generation *)
     Next,Previous: ^Variable; (* Used to doubly link the LiveList *)
     END;

   VariableList = RECORD      (* A linked list of variables *)
     Head: ^Variable;
     Tail: ^VariableList;
     END;

   Status = (Live, Dead);

CONST BitBlockSize = 32;

TYPE StatusNode = RECORD
     (* Used to maintain a stack of bits using linked blocks *)
     BitCount: 1..BitBlockSize;
     BitBlock: PACKED ARRAY [1..BitBlockSize] OF Status;
     Pop: ^StatusNode;
     END;

VAR  VariableTable = ARRAY[1..VariableTableSize] OF ^Variable;
```

The Variable Table is used to associate variables with their types and a numeric index. Variable indices are assigned sequentially, starting at zero. A hash function is used to map variable names to an entry in the

VariableTable; all the variables that map to a particular entry are linked through their BucketMate fields.

```
center;
l.
TYPE Jtags =
 (NewN, SplitN, WhenN, BranchN, JoinN, RequireN, HangN, BreakN);

Jnode =
 RECORD Next, Previous, SimLink: ^Jnode; (* see below *)
  Jtag: Jtags OF

  CASE NewN:
    (NewLiveVars, NewDeadVars): VariableList;
     NewState: StringIndex);  (* assertion from NEW instruction *)

  CASE SplitN:
    (SplitEnv: EnvIndex); (* see below *)

  CASE WhenN:
    (WhenMate: ^Jnode;
     WhenNewVars: VariableList;
     Constraint: StringIndex); (* assertion from WHEN instruction *)

  CASE BranchN:
    (BranchMate,            (* used to link successors *)
     DirectDominator: ^Jnode;
     Pathname: StringIndex;  (* string from BRANCH instruction *)

  CASE JoinN:
    (JoinEnv: EnvIndex; (* see below *)
     TraverseCount,     (* used by NewBalance *)
     InDegree:          (* number of predecessors *)
       Integer);

  Case RequireN:
    (Requirement: StringIndex;  (* Assertion that is supposed to hold *)
     ErrMsg:      StringIndex); (* Message to print if it doesn't *)

  CASE HangN: ();

  CASE BreakN: ();
  END;
```

The Next and Previous fields of a J-node point to the successor and predecessors of the node. A successor of a J-node is one that occurs immediately after its predecessor in execution order. A HangN node has no successor; its successor field is nil. A split node can have more than one successor. Its Successor field points to the first element of a list of WhenN nodes that are linked through their WhenMate fields. Likewise, JoinN nodes have more than one predecessor. The Predecessor field of a JoinN node points to a list of BranchN nodes linked through their BranchMate fields.