Model-Based Diagnosis of Hardware Description Languages[*]

Markus STUMPTNER and Franz WOTAWA
Christian Doppler Laboratory for Expert Systems
Institut für Informationssysteme
Technische Universität Wien
Paniglgasse 16, A-1040 Wien, Austria, Europe
e-mail: {mst,wotawa}@dbai.tuwien.ac.at
phone: (+43) - 1 - 588 01–6127
fax: (+43) - 1 - 505 53 04

ABSTRACT

This paper discusses the use of fault models for the diagnosis of hardware designs written in the VHDL hardware description language. VHDL designs are concurrent programs consisting of hundreds of thousands of lines of source code, whose testing and debugging is a very time-consuming process. Building on an abstract representation that allows the search for errors in even the largest VHDL programs, but with strictly limited accuracy, this paper presents a more detailed approach which is currently under study for local use in situations where focusing on a small part of the program may be possible. The approach is based on assigning fault modes to VHDL statements, with the main difference to the standard hardware diagnosis approach being that fault modes are directly expressed in terms of repair operations that will presumably improve the correctness of the program. We provide a running example and discuss the implications of the representation.

1. INTRODUCTION

In this paper we describe the application of model-based diagnosis (see [14, 8, 10] for an introduction to consistency-based and [5, 7] to abductive diagnosis) to software debugging. Similar ideas have been presented before (see, e.g., [6]) but so far little effort has been spent on this domain, in particular outside the realm of pure functional or pure logic languages. Instead of considering a system model of the physical device, we use the program or, to be exact, a model of the program which is derived using the semantics of the language (see Figure 1). From this model, we derive a behavior which has to be compared with the intended behavior of the program. The results of the comparison are given by an oracle, i.e., passed to the diagnosis system from outside. If differences (so-called discrepancies) to the intended behavior are found, the computation of diagnoses is initiated. The oracle itself can be an informal or formal specification, the user, or the result of comparing the behavior of the design for particular test runs with test runs of a second program implementing the intended behavior.

Our example uses the language VHDL (Very High speed integrated circuits Hardware Description Language) due to our involvement in the DDV project aimed at increasing productivity of circuit designers. Initial studies showed that providing better debugging support, i.e., automating discrepancy detection and diagnosis, could save up to a third of the effort in the current industrial VLSI design cycle. We introduce a system model for VHDL programs that assigns different behavioral modes to individual statements. While simple VHDL models (such as described in [9]) can be used for truly large VHDL programs (about 6 MB source code), the new model permits the explanation of faults by presenting their corrections, and therefore provides better differentiation between diagnosis candidates.

VHDL is an Ada-like programming language providing constructs for sequential, concurrent statements and recursive functions [16]. To avoid problems regarding the halting problem, we restrict our view to programs written at the register transfer level (RTL). RTL programs represent the last stage of a design before it is automatically transformed ("synthesized") into gate level programs and subsequently into real hardware. To be usable as input for synthesizing tools, RTL programs must satisfy certain simplifying restrictions, which we will point out later. A RTL program is larger and requires more simulation time in test runs than any earlier versions of the design.

Section 2 shows a VHDL example and the basic ideas behind our work, Section 3 discusses the representation of fault modes in a system description of VHDL programs,

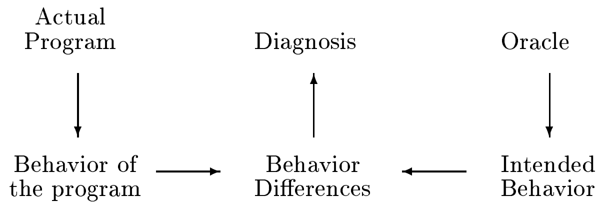

Figure 1: Program Diagnosis

*This work was partially supported by Siemens Austria, under project grant DDV GR 21/96106/4, and the Christian Doppler Laboratory for Expert Systems.

and Section 5 gives an example of the more detailed approach. The conclusion lists some current research activities.

2. A VHDL EXAMPLE

The abstract representation used in [9] for diagnosing faults in VHDL programs can in certain cases derive a large number of unnecessary diagnosis candidates. The goal of this paper is to present a new program model which provides a more discerning view of VHDL execution. We present the underlying concepts through a simple example. Consider the following VHDL program, implementing a simple RAM cell.

```
entity MEM is
     port (
          signal clk, de, rw :  in std_logic;
          signal data :  inout std_logic_vector );
end MEM;
architecture BEHAV of MEM is
begin
     MEM_UNIT: process
          variable mem:
               std_logic_vector(1 to data'range);
     begin
          if clk = '1' and de = '1' then
               if rw = '1' then
                    mem := data;
               else
                    data <= mem;
               end if;
          end if;
          wait on clk;
     end process;
end BEHAV;
```

This program makes use of VHDL *entity* and *architecture* declarations. Entities define components of a VHDL program and describe their interface. Architectures define the behavior of entities. Different architectures can be associated with different instances of the same entity. Architectures consists of concurrent statements such as the above `process` statement. Statements used inside a process are executed in sequence. The explanation of the program above is as follows: On every rising cycle of the clock `clk`, when data enable `de` is high, read (`rw = '0'`) or write cycles (`rw = '1'`) can be performed. The process then waits until an event on `clk` occurs.

Assume further that we add an encryption and decryption function to the program leading to the modified `BEHAV_2` architecture:

```
...
     mem := encrypt(data);
else
     data <= decrypt(mem);
end if;
...
```

Suppose that instead of using the correct `decrypt_new` function, the wrong `decrypt` function has been used, resulting in incorrect behavior. An approach abstracting over individual statement semantics [9] will compute the following possible diagnoses:

[`if clk = '1' and de = '1' then ...end if;`],

[`if rw = '1' then ...else ...end if;`],

[`mem := encrypt(data);`],

[`data <= decrypt(mem);`].

This is not satisfactory, and we need a more detailed model to deal with such cases.

3. FAULT MODELS FOR DIAGNOSIS

To avoid computation of irrelevant diagnoses, we have to extend the system description. In hardware diagnosis, fault models are used to partition the search space, limit diagnosis candidates (if certain observations cannot be produced by possible fault combinations), and guide search by associated probabilities. In diagnosing software, the probability of an error cannot be judged simply by the type of statement. Instead, we propose using the amount of repair (that is required for the program to perform as intended) as a measure for the probability of a diagnosis. We therefore specify all relevant behavior modes for different statement types. For the sake of simplicity, we concentrate on VHDL sequential statements. All ideas and propositions can be naturally extended to concurrent statements as well.

In contrast, the use of a system model that only describes the correct behavior while leaving the incorrect behavior unknown (as described in [14, 8]), does not lead to diagnosis improvements, because of the 'hierarchical' structure of conditional expressions. In addition, the problems which are described in [11] for diagnosing sequential circuits arise for VHDL programs as well: Effective discrimination between possible error occurrences in systems with state is very difficult.

1. *General Statements:* For all statement types mentioned above, the following fault modes are relevant:

   Correct behavior (*ok*) Correct behavior causes the computation of the statement behavior using the program semantics as given.

   Spurious statement (Remove statement) (*remove*) The effect of removing a statement is to leave variable and signal values unchanged during execution.

   Wrong statement (Replace statement by one or more new statements) (*replace*) Unfortunately, the task of finding correct replacements for a complete statement in general amounts to nothing short of automatic programming [17]. We therefore have to artificially restrict the search space which is done by limiting repair size.

Undefined fault mode (*undef*) The undefined failure mode is only used if all other modes cannot explain the misbehavior. It does not try to explain variable or signal values, and would cover cases where the program contains deep-seated logical errors.

2. *Conditional Statement:* The conditional statement can fail using the following special mode:

   Wrong condition (*cond*) This mode specifies that the condition of the conditional statement is wrong.

3. *Signal and Variable Assignment:* These two assignment types largely agree in the possible modes.

   Wrong target (*target*) If a wrong target is assumed, we can check it by inserting a new one. Only defined variables (signals for signal assignments) can be used for target replacement.

   Wrong expression (*expr*) A wrong expression implies that there must be a correct one. Checking this mode is done by replacing the original expression by a new one. The type of the result must be the same.

   Wrong time expression (*time*) This mode exists for signal assignment statements only and indicates an error in the `after` clause.

4. A FORMAL MODEL

We describe the syntax and semantics of a subset of VHDL which is relevant for our purpose. Figure 4 gives a brief overview of the syntax. To simplify our considerations we assume that loop statements and recursive functions do not exist. Note that these assumptions hold for synthesizable RTL programs.

The evaluation of process statements which are used to describe Architectures occurs in parallel. Inside processes, sequential statements are used.

In the rest of this section we give the semantics of the VHDL subset. For the sake of simplicity we ignore time and concentrate only on the execution and fault diagnosis of sequential statements. Within a block of sequential statements, signal and variable values are changed. We represent the association between variables (and signals) and their variables through *environments*. A changed value results in the creation of a new environment.

$$EnvSet = \{ {}^{S}_{V} \mid S \in Signals, V \in domain(S), \not\exists {}^{S}_{V'} \in EnvSet : V' \neq V \}$$

For our purposes we assume that environments do not necessarily store values for all variables and signals, i.e., every environment E is a subset or equal to $EnvSet$. If the exact value has not been computed or cannot be computed due to incorrectness assumptions, undefined values are assumed for the missing variables.

The operation [ ] allows to modify environment items, and is defined as follows:

```
entity_declaration ::=
      entity id is
            [ port (
                  interface_declarations ) ; ]
      end id ;
interface_declarations ::=
      interface_declaration int_decl_rest
int_decl_rest ::=
      ; interface_declaration int_decl_rest
      | ε
interface_declaration ::=
      signal id_list : [ mode ] type_id [ := expr ]
mode ::=
      in | out | inout
architecture_declaration ::=
      architecture id of id is
            architecture_declarations
      begin
            concurrent_statements
      end id ;
concurrent_statements ::=
      process_statement concurrent_statements
      | ε
process_statement ::=
      id : process
            process_declarations
      begin
            sequential_statements
            wait_statement ;
      end process id ;
sequential_statements ::=
      sequential_statement ; sequential_statements
      | ε
sequential_statement ::=
      signal_assignment
      | variable_assignment
      | if_statement
      | wait_statement
signal_assignment ::=
      id <= [ transport ] expr after expr
variable_assignment ::=
      id := expr
if_statement ::=
      if expr then
            sequential_statements
      else
            sequential_statements
      end if
wait_statement ::=
      wait wait_statement_rest
wait_statement_rest ::=
      on id_list
      | for expr
      | until expr
id_list ::=
      id id_list_e
id_list_e ::=
      , id id_list_e
      | ε
...
```

Figure 2: Syntax of a VHDL Subset

$Env[{}^{S}_{V}] = \left\{ {}^{X}_{Y} \,|\, {}^{X}_{Y} \in Env \wedge X \neq S \right\} \cup \left\{ {}^{S}_{V} \right\}$

where Env is an environment, S a signal or variable, and V the new value.

The correct behavior of statements can be described as follows:

- $S =$ "T `<=` E `after` TE"
  $ok(S) \Rightarrow$
  $\big(eval(S, Env) = Env[{}^{T}_{eval_E(E,Env)}]\big)$
- $S =$ "T `:=` E"
  $ok(S) \Rightarrow$
  $\big(eval(S, Env) = Env[{}^{T}_{eval_E(E,Env)}]\big)$
- $S =$ "`if` C `then` St_1 `else` St_2"
  $ok(S) \Rightarrow$
  $\big(eval_E(C, Env) = true \rightarrow$
  $eval(S, Env) = eval(St_1, Env)\big)$
  $ok(S) \Rightarrow$
  $\big(eval_E(C, Env) = false \rightarrow$
  $eval(S, Env) = eval(St_2, Env)\big)$
- $S =$ "$St_1 \dots St_n$"
  $eval(\{St_1, \dots, St_n\}, Env) =$
  $eval(\{St_2, \dots, St_n\}, eval(St_1, Env))$
  $eval(\{St\}, Env) = eval(St, Env)$

Undefined incorrect behavior is expressed by:

- $S =$ "T `<=` E `after` TE"
  $undef(S) \Rightarrow$
  $\big(eval(S, Env) = remove(Env, Env[{}^{T}_{V}])\big)$
- $S =$ "T `:=` E"
  $undef(S) \Rightarrow$
  $\big(eval(S, Env) = remove(Env, Env[{}^{T}_{V}])\big)$
- $S =$ "`if` C `then` St_1 `else` St_2"
  $undef(S) \Rightarrow$
  $\big(eval_E(C, Env) = true \rightarrow$
  $eval(S, Env) = remove(Env, eval(St_1, Env))\big)$
  $undef(S) \Rightarrow$
  $\big(eval_E(C, Env) = false) \rightarrow$
  $eval(S, Env) = remove(Env, eval(St_2, Env))\big)$

The function $remove$ deletes all variable (or signal) entries from the second parameter which differ from the corresponding entries of the first parameter.

The other fault modes can be formalized by:

- $S =$ "T `<=` E `after` TE"
  $target_{T'}(S) \Rightarrow$
  $\big(eval(S, Env) = Env[{}^{T'}_{eval_E(E,Env)}]\big)$
  $expression_{E'}(S) \Rightarrow$
  $\big(eval(S, Env) = Env[{}^{T}_{eval(E',Env)}]\big)$
  The fault mode description for wrong time expression is omitted.
- $S =$ "T `:=` E"
  $target_{T'}(S) \Rightarrow$
  $\big(eval(S, Env) = Env[{}^{T'}_{eval_E(E,Env)}]\big)$
  $expression_{E'}(S) \Rightarrow$
  $\big(eval(S, Env) = Env[{}^{T}_{eval_E(E',Env)}]\big)$
- $S =$ "`if` C `then` St_1 `else` St_2"
  $cond(S) \Rightarrow$
  $\big(eval_E(C, Env) = true \rightarrow$
  $eval(S, Env) = eval(St_2, Env)\big)$
  $cond(S) \Rightarrow$
  $\big(eval_E(C, Env) = false \rightarrow$
  $eval(S, Env) = eval(St_1, Env)\big)$
- General fault modes for statements S
  $remove(S) \Rightarrow$
  $eval(S, Env) = Env$

The description of statement replacements is omitted.

Since `wait` statements do not influence the computed values, evalution is simple and can be expressed by the fact

$eval(\texttt{wait}\dots, Env) = Env.$

Apart from the system description, observations are required to diagnose misbehaviors. In our context, observations are represented by environments. To prove that a program does not contradict the given observations we have to use the evaluation predicate $eval$. In addition, we introduce the rule

$eval(S, Env) \neq Env_{expected} \rightarrow \bot$

saying that the program should behave as expected. The $\neq$ predicate is defined as follows:

$$Env' = Env'' \Leftrightarrow$$
$$(\forall {}^{S}_{V} \in Env' : \exists {}^{S}_{V'} inEnv'' : V = V') \wedge$$
$$(\forall {}^{S}_{V} \in Env'' : \exists {}^{S}_{V'} inEnv' : V = V').$$

$$Env' \neq Env'' \Leftrightarrow \neg(Env' = Env'').$$

If nothing is known about the value of a variable or signal we cannot derive a contradiction.

An example should clarify the basic concepts. Consider the implementation of the `BEHAV_2` architecture from our running example. For easier referencing, we have numbered the statements.

```
...
begin
```

```
        if clk = '1' and de = '1' then - -St1
            if rw = '1' then - - St2
                mem := encrypt(data); - -St3
            else
                data <= decrypt(mem); - - St4
            end if;
        end if;
        wait on clk; - -St5
    end process;
end BEHAV_2;
```

Assume further that we have the input environment

$Env = \{ {}^{clk}_{'1'}, {}^{de}_{'1'}, {}^{rw}_{'0'}, {}^{mem}_{'1011'}, {}^{data}_{'UUUU'} \}$

and expect the output environment

$Env' = \{ {}^{clk}_{'1'}, {}^{de}_{'1'}, {}^{rw}_{'0'}, {}^{mem}_{'1011'}, {}^{data}_{'0100'} \}.$

Now assume the incorrect value `'0101'` was computed for signal `data` (because `decrypt` was used instead of `decrypt_new`). We now show how a contradiction can be derived if all statements are correct, i.e., $\forall St \in \{St1, \ldots, St5\} : ok(St)$. First, assume that

$eval(St1, \{ {}^{clk}_{'1'}, {}^{de}_{'1'}, {}^{rw}_{'0'}, {}^{mem}_{'1011'}, {}^{data}_{'UUUU'} \}) = \{ \ldots, {}^{data}_{'0101'} \}$

can be derived. Since

$eval($`clk = '1' and de = '1'`$, Env) = true$

holds, it follows that $eval(St2, Env) = \{ \ldots, {}^{data}_{'0101'} \}$, while $eval($`rw = '1'`$, Env) = false$ implies that $eval(St4, Env) = \{ \ldots, {}^{data}_{'0101'} \}$ holds. From the contradiction, it follows that the assumption that all components behave correctly must be relaxed and one or several components must behave incorrectly, resulting in possible diagnoses.

Applying fault models to our example results in a more focused diagnosis set. For example, the diagnosis [`if clk = '1' and de = '1' then ...end if;`] can be removed, because assuming *cond* mode for the statement does not correct the program. The *undef* mode would be consistent with the fault, but, as stated above, would have a very small probability. The new diagnosis set of (mode,statement,repair) elements includes only the expected diagnosis [*expr*,`data <= decrypt(mem);`, `decrypt(mem)` $\Leftarrow$ `decrypt_new(mem);`].

Diagnoses can only be computed after a set of fault modes has been determined. As mentioned above this set must be restricted because it is infinite in general. To do this we introduce a heuristic limitation to fault modes which change only a small part of the program. A set of modes for the decryption example could contain

$cond(St1), cond(St2), expr_{\texttt{decrypt\_2(mem)}}(St4)$

and also

$expr_{\texttt{clk = '1' or de = '1'}}(St1),$

$expr_{\texttt{decrypt(mem) and '1110'}}(St4), \ldots$

It should be clear that some modes induce more complex changes than others. We use the difference between the original expression (statement) and the new one given by the mode as a rating for ordering modes. A iterative deepening algorithm first searches for (single) diagnoses using the modes with the smallest difference and then increases the difference bound if no diagnosis has been found.

5. SYSTEM DESCRIPTION USING FAULT MODELS

As shown in the example, the set of fault modes provides a possibility to partition the search space in a meaningful way according to VHDL semantics. Unfortunately, computational complexity is significantly higher than with the abstract representation of [9].

- Finding all possible replacements is intractable. The search space must be limited by heuristic means, e.g., by assuming that the intended program is only a (very) small variant of the original one. This particular assumption is what marks the difference to automatic programming and allows the application of the approach to real world examples.
- Attempts to assign certain fault modes could be delayed during diagnosis, e.g., by initially looking only at those names with small differences to a possibly incorrect one, or using fault probabilities to choose initial modes. Various other techniques can be envisioned to guide search in this manner. On the other hand search is restricted to the objects declared within a program. Replacing functions, signals and variables by other functions, signals and variables is only possible if all such objects are within the declaration scope and if they are type compatible. In our example above, no *target* fault can be assumed, because there is no second output signal available.
- It is possible that changing the program behavior may lead to infinite loops. This is no problem for diagnosing synthesizable VHDL programs where recursive function and loops with variable parameters are not allowed. In unrestricted programs, heuristic judgments on program termination or computational restrictions must be used to avoid this problem.

6. DISCUSSION

We are currently developing a model for VHDL semantics using first order logic that is suitable for the definition of fault models.

Writing fault models for software diagnosis is a good way of explaining faulty behavior by presenting solutions. However, several problems regarding the computational complexity remain to be solved (see above). Most of them can be alleviated by introducing heuristic assumptions about the application domain which place restrictions on the set of possible replacements. Examples include

- that a replacement may only refer to declared signals and variables,

- that replacements must be type compatible (since a type error would correspond to hidden multiple errors)
- most importantly, the assumption that the intended program is only a (very) small variant of the original one. While this does not exactly place a limit on the search space, it provides an ordering that allows the pruning of many diagnoses of lesser probability.

The notion of intelligent debugging aids has been proposed repeatedly in the literature. What distinguishes our approach from most others is the attempt to rely as little as possible on specifications external to the program. The observations used in [9] are exclusively derived from the output of VHDL simulation runs, and in this case, we attempt to limit ourselves as far as possible to comparisons with individual output values (tests). Other work [12, 13] has usually assumed the existence of a complete analyzable specification or even a similar implementation to guide search for the source of the errors. While this makes it certainly easier to pinpoint individual errors, the research described shows these approaches being used exclusively relatively small (even if nontrivial) examples. The Algorithmic Debugging approach of Shapiro[15] also attempts to find the source of errors by a derivation process about the properties of a program. However, the approach is limited by its strong reliance on the semantic properties of pure logic programs, and the necessity for a high degree of user interaction (see [6] for a discussion).

A different approach to determining the correctness of hardware designs that has aroused considerable interest in recent years is the use of formal verification, using model checking [3] or other techniques [2]. Disadvantages of this approach again include the necessity to provide a separate formal specification, which requires considerable design effort in itself and, for large systems, is a source of design errors in its own right. Just as important, the practical computational complexity of the verification task is such that it precludes its use for large systems or designs implementing complex algorithms, though the use of sophisticated abstraction techniques has led to increases in capability in the last years [4].

7. CONCLUSION

This paper describes a partial semantic model for VHDL using first order logic that is amenable to the use of fault models for model-based diagnosis. This research is part of a larger project which has so far led to the implementation of a more abstract system model [9] that is currently being tested in actual use. While its simplicity enables its use for life-size VHDL programs, its discriminating power is limited with regard to programs that have a complex internal topology.

The main distinction of the model-based approach with regard to other work aimed at intelligent software debugging aids lies in the possibility to largely avoid the use of external specifications that place an additional training and design load on the actual designers. A VHDL parser for full VHDL has been implemented and can be used for automatic generation of the system model from VHDL source code. The use of fault models is one of several paths being currently pursued with the ultimate purpose of providing good discrimination for diagnoses in limited parts of larger VHDL programs.

8. REFERENCES

[1] T. Albrecht, G. Friedrich, M. Stumptner, and F. Wotawa. WFCOMP and VHDLDIAG: Two Tools for Utilizing the Hardware Design Cycle. Technical Report DBAI-MBD-TR-95-04, Technische Universität Wien, June 1995.

[2] R. Boyer and J. Moore. *A computational logic handbook*. Academic Press, 1988.

[3] E. Clarke, E. Emerson, and A. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Database Systems*, 8(2):244–263, Apr. 1986.

[4] E. M. Clarke, O. Grumberg, and D. E. Long. Model Checking and Abstraction. *ACM Transactions on Database Systems*, 16(5):1512–1542, Sept. 1994.

[5] L. Console, D. T. Dupré, and P. Torasso. A theory of diagnosis for incomplete causal models. In *Proc. IJCAI*, pages 1311–1317, Detroit, Aug. 1989. Morgan Kaufmann Publishers, Inc.

[6] L. Console, G. Friedrich, and D. T. Dupré. Model-based diagnosis meets error diagnosis in logic programs. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1494–1499, Chambery, Aug. 1993. Morgan Kaufmann Publishers, Inc.

[7] L. Console and P. Torasso. Integrating models of correct behavior into abductive diagnosis. In *Proc. ECAI*, pages 160–166, Stockholm, Aug. 1990. Pitman Publishing.

[8] J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[9] G. Friedrich, M. Stumptner, and F. Wotawa. Model-based diagnosis of hardware designs. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, Budapest, Aug. 1996.

[10] R. Greiner, B. A. Smith, and R. W. Wilkerson. A correction to the algorithm in Reiter's theory of diagnosis. *Artificial Intelligence*, 41(1):79–88, 1989.

[11] W. C. Hamscher and R. Davis. Diagnosing circuits with state - an inherently underconstrained problem. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 276–282. Morgan Kaufmann Publishers, Inc., 1984.

[12] W. L. Johnson. *Intention-Based Diagnosis of Novice Programming Errors*. Pitman Publishing, 1986.

[13] W. R. Murray. *Automatic Program Debugging for Intelligent Tutoring Systems*. Pitman Publishing, 1988.

[14] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[15] E. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, Massachusetts, 1983.

[16] IEEE Standard VHDL Language Reference Manual LRM Std 1076-1987, 1988.

[17] R. J. Waldinger and R. C. T. Lee. PROW: A step toward automatic program writing. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, Washington, DC, May 1969.