A Model-Based Approach to Software Debugging

Markus Stumptner and Franz Wotawa
Institut für Informationssysteme
Technische Universität Wien
Paniglgasse 16, A-1040 Vienna, Austria, Europe
Phone: (+43) - 1 - 588 01–6127
Fax: (+43) - 1 - 505 53 04
E-Mail: {mst,wotawa}@dbai.tuwien.ac.at

Abstract: This paper introduces the application of model-based diagnosis to the debugging of functional programming languages. We describe the generation of a logic system description out of the evaluation trace of a program and discuss the differences in perspective that arise out of the lack of a structurally complete specification for the program. We present two different approaches for diagnosing faults that result in incorrect executions. The first (simpler and quicker) approach uses a fixed system description based on the evaluation trace, and the second (more accurate) approach incorporates the search for a repair that will correct the fault, expressed in terms of fault modes corresponding to possible repairs. We describe the algorithm for computing diagnoses based on repairs, discuss how the faster algorithm can be used for focusing search before the second algorithm computes replacements, and finally present some example results.

Keywords: program debugging, repair, fault modes

1 Introduction

Despite the long history of model-based diagnosis, its use has so far been mostly restricted to the search for hardware faults. First attempts to extend the paradigm towards the search for errors in software artifacts (i.e., programs) were made in (Console, Friedrich, & Dupré 1993) (see also (Bond & Pagurek 1994; Bond 1994)). The idea was to take a logic program and remove, add, or replace whole clauses with the goal of arriving at a program that would produce correct answers to certain test queries. This work took the shape of a thought experiment; it utilized the declarative nature and clear semantics of logic programs, and it was not used on large programs. The research described in this paper has its origin in the DDV project which uses MBD for diagnosing hardware designs written in the VHDL specification language (Friedrich, Stumptner, & Wotawa 1995). VHDL designs basically are very large programs written in an imperative programming language with added constructs to represent concurrency in hardware devices. The initial approach chosen for diagnosis in the DDV project was that of using a strongly abstract representation so that effective search for diagnoses in circuit designs with up to 1,000,000 LOC is possible. The components to be diagnosed are the concurrent statements which make up the bodies of VHDL processes. The system description is based on representing functional dependencies between signals in the design.

However, the abstract representation naturally comes with the disadvantage that discrimination between diagnoses is limited. It seems advantageous to use a more detailed representation, including the representation of individual signal and variable values, of the semantics of the VHDL language for a more effective distinction between possible errors. As an initial step, we decided to define a simple functional language and examine the application of MBD to this language. The results can be used in the context of the VHDLDIAG tool for improved discrimination between diagnoses, but also apply to functional languages in general. Our approach is based on the standard notions of consistency-based diagnosis (see (de Kleer & Williams 1987; Reiter 1987) for an introduction). This is a deviation from the approach of (Console, Friedrich, & Dupré 1993), which was closer to abductive diagnosis. (See (Console & Torasso 1990) for an concise comparison of both.) In addition, considering only pure logical languages in diagnosis helps finding an appropriate model since the program semantics allow a trivial translation into a logic-based system description.

A fundamental difference between program debugging and hardware diagnosis lies in the nature of the faults in the two areas. One of the assumptions made in hardware diagnosis is that the structure of the artifact to be diagnosed is known and that the system description provides a model of the correctly functioning artifact.

When dealing with a program error, the situation is different.

First, no independent system description depicting the correct program will be available. Even if a formal specification of the program exists, the axioms of the specification will have little or no structural commonality with the procedures and processes of the actual program.

Second, whereas hardware malfunctions are usually modelled as the production of incorrect output values for the system, a program bug may also manifest itself by the fact that the program does not terminate on a given input. In this case, no output values are given. We do not currently deal with this case in detail (which is also justifiable in the VHDLDIAG context, as the most common type of VHDL programs, Register-Transfer-Level programs, do not contain recursive expressions or loops so that expression evaluation in such programs always terminates).

In hardware diagnosis, the system description is used to derive statements about the correct functioning of the system, and these are compared to observations which report the actual (incorrect) functioning. In debugging programs, the output of the program is incorrect because it was incorrectly designed. Therefore, the system description (the model of the actual artifact) is by necessity a model of the only existing version of the program, which is the *incorrect* version which delivered the bug, and the observations used in diagnosis (that the system description can be compared to) are the expected correct outputs of a given individual (or set of) test cases.

Since the structure of the system is not necessarily correct as described, the notion of a fault takes on a more general meaning. In particular, structural faults are possible. In addition, it is usually not possible to assign empirical probabilities to individual kinds of errors. Instead, possible error candidates are ranked by the size of the program change needed to correct them. This viewpoint was initially introduced in (Console, Friedrich, & Dupré 1993), with clauses taken as units of replacement. In a functional language, the logical candidates for replacement are expressions, which results in a lower granularity and hopefully more effective results.

The main contributions of this paper can be summarized as follows: We define the simple functional language $\mathcal{EXP}$ and describe how to map $\mathcal{EXP}$ programs to a system description usable for diagnosis. This can happen at different levels of abstraction. In particular, we discuss the introduction of fault modes and replacements and their importance for focusing diagnosis.

2 Introducing $\mathcal{EXP}$

Expressions in $\mathcal{EXP}$ include terms (variables, constants and function calls), conditionals, and user defined functions. We assume that all constants and built-in functions are related to specific types such as *integer* or *string*. $\mathcal{EXP}$ is defined on a set of variable symbols VS, constant symbols A, (built-in) function names FB, and user-defined function names FV (i.e., function-valued variables). Every function name has an arity associated with it. Function calls are denoted by $f(e_1, \ldots, e_n)$ where $f \in FB \cup FV$ and conditionals by $cond(e_1, e_2, e_3)$.

We start the discussion of $\mathcal{EXP}$ semantics by defining an environment mechanism that allows us to express the association between variables and their values as well as functions and their bodies.

Definition 2.1 (Environments) A variable environment *is a function* $\iota : VS \mapsto VALUE$. A function environment *is a function* $\delta : FV \mapsto \mathcal{EXP}$ *associating expressions with function symbols. We denote the set of all variable environments by* $VENV$*, the set of all variable environments by* $FENV$*.*

Constant symbols syntactically represent constants. We write $const(a)$ to denote the constant represented by the constant symbol a. Also, we use $vars(f)$ to denote the variables used in the body of a user-defined function f. Finally, for simplicity, we consider the function environment to be constant throughout the program.

The semantics of $\mathcal{EXP}$ are given by the evaluation functions $eval : \mathcal{EXP} \times VENV \times FENV \mapsto VALUE$ which describes the result of the evaluation of an expression in a given variable and function environment:

1. $eval(X, \iota, \delta) = \iota(X)$ if $X \in VS$
2. $eval(X, \iota, \delta) = const(X)$ if $X \in A$.
3. $eval(cond(x_1, x_2, x_3), \iota, \delta) = \begin{cases} eval(x_2, \iota, \delta) & \text{if } eval(x_1, \iota, \delta) = true \\ eval(x_3, \iota, \delta) & \text{otherwise} \end{cases}$
4. $eval(f(x_1, \ldots, x_n), \iota, \delta) = f(eval(x_1, \iota, \delta), \ldots, eval(x_n, \iota, \delta))$ if $f \in FB$.
5. $eval(F(x_1, \ldots, x_n), \iota, \delta) = eval(\delta(F), \iota', \delta)$ where $\iota'(v_i) = eval(x_i)$ and $v_i \in vars(F)$ and $\iota'(v) = \iota(v)$ if $v \notin vars(F)$.

We define a *type* as triple consisting of the domain, a set of (built-in) functions, and a set of predicates defined on the domain. Usually types are indicated by their domain. From now we use the type $\mathbb{N}$ with the domain $\mathbb{N}$, the built-in functions *inc*, *dec*, *plus*, *minus*, *div*, *mult*, and the predicates (i.e., functions of result type boolean) *greater*, *ge* (for greater or equal), and *equal*, with the expected semantics. A *program* Π is a tuple (T, δ) consists of a set of types T, together with user-defined functions δ building on these types.

A query given to such a program Π is an $\mathcal{EXP}$ expression which answers the value given back by the evaluation process using the *eval* function.

Example 1: The following $\mathcal{EXP}$ program is supposed to implement the multiplication function, but is incorrect.

```
myMult(x1,x2) =
     cond(ge(x1,2),
          plus(x2, myMult( minus(x1,1), x2))
          0)
```

The query $eval(myMult, 2, 3)$ returns the incorrect result 3. $myMult$ can be corrected by writting `ge(x1,1)` instead of `ge(x1,2)`. $\square$

A set of test cases for a program corresponds to a set of queries with the expected result for each query. If the value returned by a query is not the expected one (such as in the previous example), a misbehavior has occurred. This could in theory be the result of an incorrect query, an incorrect implementation of a built-in function, or a bug in a user-defined function. Clearly, the last case is what we are interested in. The point of interest becomes the identification of the part(s) of the program definition which cause the error. This requires the ability to identify and distinguish expressions.

Expressions can be identical (denoted by $\equiv$), syntactically equivalent ($=$), or evaluation equivalent ($=_E$). Identity is unique for every expression in the program code, e.g., two occurrences of the expression $ge(x1, 1)$ at two places in the program would have a different identity even though they are syntactically equivalent. Identity implies equality and equality implies evaluation equality.

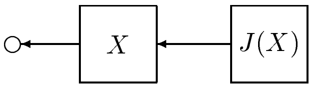
To allow easy differentiation between expressions we assume that every expression has a unique identifier and there is a function $\kappa : \mathcal{EXP} \mapsto \mathbb{N}$ returning the identifier of an expression. Obviously: $e1 \equiv e2 \Leftrightarrow \kappa(i1) = \kappa(i2)$.

Next, we characterize which expressions are used (syntactically) in the computation of another expression. We say that e is a subexpression of an expression e' (denoted by $e \subseteq e'$) if e occurs in e' or if both are identical. We further introduce a function ϵ returning all expressions used in a program Π ($\epsilon(\Pi) = \{e | F \in FV \wedge e \subseteq \delta(F)\}$).

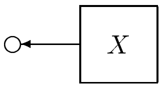
During the evaluation of an expression multiple calls to the *eval* function will occur. An evaluation trace is a graph representing the interdependencies between calls. It will be the basis for defining a system description in Section 4.

Definition 2.2 (Evaluation Trace Function) *Let Π be a program, q a query, and J a function mapping vertices to sets of variables. The evaluation trace function $evalT(q, \iota, \delta, n, J)$ for a function call $eval(q, \iota, \delta)$ defines a tuple (V, A, tn, nn, val), where V and A are the vertex and arc sets of the evaluation graph, respectively, tn is the top node of the graph, nn is the argument n incremented by the number of nodes in the graph, and $val = eval(q, \iota, \delta)$. The tuple (V, A, tn, nn, val) is defined as follows for the different kinds of expression:*[1]

Variables:
$evalT(X, \iota, \delta, n, J) =$
$(\{n\}, \{(J(X), n)\}, \{n\}, n + 1, \iota(X))$ *if* $X \in VS$



Constants:
$evalT(X, \iota, \delta, n, J) =$
$(\{n\}, \{\}, \{n\}, n + 1, const(X))$ *if* $X \in A$.



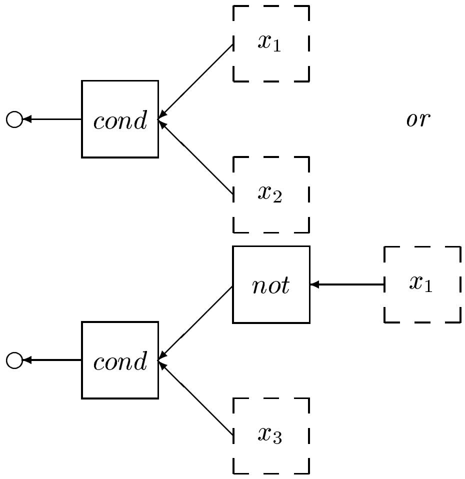
Conditional expressions: *The trace contains the evaluation of the expression, and the evaluation of the chosen path.*

Given a conditional $cond(x_1, x_2, x_3)$ where $(V_1, A_1, TN_1, NN_1, Val_1) = evalT(x_1, \iota, \delta, N + 1, J)$ is the trace of the evaluation of x_1, then

$evalT(cond(x_1, x_2, x_3), \iota, \delta, N, J) =$
$(V, A, \{N\}, NN, Val)$ *where* $V = \{N\} \cup V_1 \cup V'$, $A = \bigcup_{x \in TN'} \{(N, x)\} \cup \{(N, TN_1)\} \cup A_1 \cup A'$,
$NN = NN' + 1$, *and* $Val = Val'$

[1] In the figures below, dashed boxes represent vertices which are the roots of subgraphs not shown in the figure.

If $Val_1 = true$*, then* $(V', A', TN', NN', Val') = evalT(x_2, \iota, \delta, NN_1, J)$*,*
otherwise
$(V', A', TN', NN', Val') = evalT(x_3, \iota, \delta, NN_1, J)$

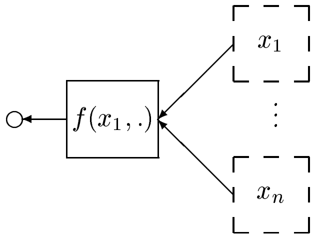


Built-in functions: *The trace contains the evaluation of the arguments and the application of the function to the arguments.*

$evalT(f(x_1, \ldots, x_n), \iota, \delta, N, J) = (\{N\} \cup \bigcup_{i=1}^{n} V_i, \bigcup_{i=1}^{n}\{(N, TN_i)\}, NN_n, f(Val_1, \ldots, Val_n))$*, where*

$(V_1, A_1, TN_1, NN_1, Val_1) = evalT(x_1, \iota, \delta, N+1, J)$

$(V_i, A_i, TN_i, NN_i, Val_i) = evalT(x_i, \iota, \delta, NN_{i-1}, J)$ *for* $i \in \{1, \ldots, n\}$.



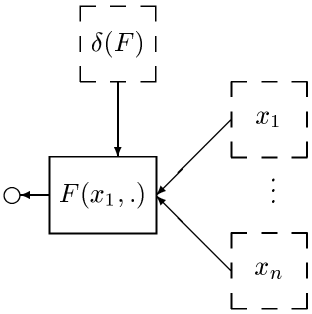
User-defined Functions: *Just as for built-in functions, but the trace also contains the evaluation of the function name (i.e., the access to the function body).*

$evalT(F(x_1, \ldots, x_n), \iota, \delta, N, J) =$
$(V, A, \{N\}, NN, Val)$ *where* $V = \{N\} \cup \bigcup_{i=1}^{n}\{V_1\} \cup V'$, $A = \{(N, TN')\} \cup \bigcup_{i=1}^{n}\{(N, TN_i)\}$, $NN = NN_n$, *and* $Val = Val'$.

$(V', A', TN', NN', Val') =$
$evalT(\delta(F), \iota', \delta, NN_n, J')$ *where*
$I'(v_i) = Val_i, J'(v_i) = TN_i$ *if* $v_i \in vars(F)$ *and* $\iota'(v) = \iota(v), J'(v) = J(v)$ *if* $v \notin vars(F)$

$(V_1, A_1, TN_1, NN_1, Val_1) = evalT(x_1, \iota, \delta, N+1, J)$

$(V_i, A_i, TN_i, NN_i, Val_i) = evalT(x_i, \iota, \delta, NN_{i-1}, J)$ *for* $i \in \{1, \ldots, n\}$

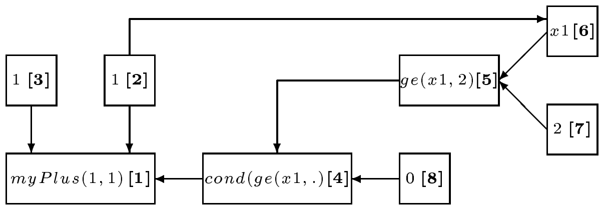


As can be seen from the definition, every time a user-defined function is called, the graph representation of those parts of its body that are actually evaluated (subject to conditional expressions) is added to the evaluation trace graph. The arcs of the graph correspond to the direction of the flow of values in the evaluator, i.e., there is an arc from a subexpression x to the expression that uses the value computed by evaluating x.

We also assume the existence of a function *expr* mapping every vertex of the graph to an element of $\mathcal{EXP}$, a function *varEnv* associating a variable environment with every vertex, and a function *vertex* as the inverse of *expr*, i.e., $vertex(e)$ returns the set of vertices v such that $expr(v) = e$. Note that $vertex(e)$ will be a singleton set unless the evaluation contains recursive calls or a user-defined function is called several times with the same arguments.

Definition 2.3 (Evaluation Trace) *Let* Π *be a program,* q *a query, and* J *a function mapping vertices to sets of variables. The evaluation trace for the evaluation* $eval(q, \iota, \delta)$*, denoted by* $evalTrace(q, \iota, \delta)$*, is the graph* (V, A) *where* $(V, A, TN, NN, Val) = evalT(q, \iota, \delta, 1, J)$.

Example 2: We pose the query $myMult(1, 1)$ for the program from example 2, resulting in the evaluation trace below. The expression identifier is given in braces ([ ]).



□

Theorem 2.1 *Let* Π *be a program and* q *a query. The evaluation trace* $\mathcal{G}$ *of* q *is acyclic.*

It is clear that a program only halts on a query if the associated evaluation trace is finite.

3 Diagnosis

We now have the tools we need for computing diagnoses given an $\mathcal{EXP}$ program and a test case. We follow the basic definitions of (Reiter 1987; Greiner, Smith, & Wilkerson 1989; de Kleer, Mackworth, & Reiter 1990). The basic parts of the program to be diagnosed are the expressions and subexpressions contained in its code. It is the expressions which correspond to the components in the traditional MBD view. The diagnosis problem can therefore be formulated as the search for a set of expressions whose "malfunction" (i.e., incorrectness) will explain the incorrect results produced by the program.

Definition 3.1 (Diagnosis Problem) *A diagnosis problem is a triple* $(SD, COMP, OBS)$ *where* SD *is the system description of the system to be diagnosed,* $COMP$ *is the set of components in the system, and* OBS *is a finite set of observations.*

Returning to the discussion of software errors from Section 1, we note that since a misbehavior is the result of an incorrect part of the program, it follows that a natural way of classifying errors is to look at how the program must be altered to remove the error. Thus, for example, omitting an operator (as in writing "`i`" instead of "`i+1`" is a different kind of error from using the wrong variable (as in writing "`a[i]`" instead of "`a[j]`"). The way in which an expression has been incorrectly used in a program can therefore be likened to the fault modes of hardware diagnosis (and, in the same manner as a fault mode, a particular assumption for the "corrected" statement limits the difference in behavior between the correct and incorrect system). We therefore define the notion of component modes for a functional program (where the components are the expressions in the code of the program). The behavior of components must be defined at least in the case where a component is assumed to behave correctly, but in general, specific fault modes will also imply a specific behavior.

The set $MODES$ contains all behavior modes for components. At least the two distinct modes $undef$ and ok must be element of $MODES$. The $undef$ mode will not usually have a behavior description assigned to it, it is intended as a "catchall" mode in cases where more specific modes are found to be inapplicable.

Definition 3.2 (Mode Assignment) *A mode assignment* μ *is a set of ground facts of the form* $m(c)$ *where* $m \in MODES$ *and* $c \in COMP$*, such that no component is assigned two different modes.*

Definition 3.3 (Diagnosis)
Let $(SD, COMP, OBS)$ *be a diagnosis problem. A mode assignment* Δ *is a diagnosis* $(DIAG(\Delta))$ *iff* $SD \cup OBS \cup \Delta \cup \{ok(c) | c \in COMP \setminus \gamma(\Delta)\}$ *is consistent.*

The task of a diagnosis engine is to determine mode assignments for every component which explain the error. For the sake of brevity, we assume in this paper that OBS corresponds to the one particular output value expected from a query. To limit the search space, we will only look for minimal diagnoses in the spirit of (Reiter 1987), and define an appropriate order on mode assignments.

Definition 3.4 (Mode Assignment Subsumption)
Let MA_1, MA_2 *be two mode assignments. We say that* MA_1 *subsumes* MA_2 $(MA_1 \sqsubseteq MA_2)$ *if the following holds:*

$$\forall m(c) \in MA_1 : m(c) \in MA_2 \vee undef(c) \in MA_2$$

The above definition prefers mode assignments with smaller cardinality and mode assignments which avoid the undefined behavior mode[2].

Definition 3.5 (Minimal Diagnosis) *Let* Δ *be a diagnosis for* $(SD, COMP, OBS)$*. We say* Δ *is minimal iff there exists no diagnosis subsuming* Δ*.*

$$MIN(\Delta) \Leftrightarrow \nexists \Delta' : DIAG(\Delta') \wedge \Delta' \sqsubseteq \Delta.$$

4 Using Evaluation Traces for Diagnosis

As discussed in Section 3, we need to use knowledge about the diagnosis components and their behavior for diagnosis. The simplest approach of all is to use the evaluation graph for a given query and attempt to derive contradictions using the assumption of correctness for all program components. Vertices of the graph are viewed as diagnosis components and hence pointers to the expressions they represent. We only have to define a behavior for all components. Misbehavior of a component is interpreted to mean that the component is inappropriate at that position in the program, and either it, or a subexpression, should be altered. In principle, To do this we must take the diagnosis capability

[2] Since the $undef$ mode does not explain why a component may fail.

and the underlying fault semantics into account. Below, we describe the behavior model for each different $\mathcal{EXP}$ construct.

Variable If the variable component is said to be correct, the output value (e.g. a constant value) should be propagated to the input.

$ok(V) \rightarrow out(V) = in(V)$

Constant Since constants may be bound to variables they may have several outputs but never an input. If a constant component is correct, the value of the constant is propagated to the outputs.

$ok(C) \rightarrow out_i(C) = const(C)$

Conditional Expression If the conditional expression is correct and the first input is true, then the result of the second is to be propagated to the output.

$$ok(Cond) \wedge in_1(Cond) = true \rightarrow \\ out(Cond) = in_2(Cond)$$

Note that due to the definition of the evaluation trace, the second input may be connected to the *then* or the *else* branch of the conditional statement. If it is connected to the *else* branch, an additional *not* added to the condition expression causes the first input to be true.

Built-in Function Call Note that even a call to a built-in function could be incorrect (it might have to be replaced by a call to a different function or another kind of expression to correct the program).

$$ok(BFC) \rightarrow \\ out(BFC) = f(in_1(BFC), \ldots, in_n(BFC))$$

User-defined Function Basically the same faults as for built-in function calls can occur for user-defined functions. If the correct function is called, the behavior (as expressed by accessing the body via the function environment) has to be propagated to the output if all other inputs (related to the parameter expressions) deliver a value.

$ok(F) \wedge \bigwedge_{i=1}^{n} defined(in_i(F)) \rightarrow out(F) = in(F)$

(The predicate $defined/1$ is true if the value of the input is equal to the expected value given by the standard expression evaluation.)

The evaluation trace graph and the above defined system description gives the behavior and connectivity of a system which can be diagnosed using standard techniques. Since there are no fault models, the notion of diagnosis is purely consistency-based.

A result ignoring vertices and using the basic expressions can be easily computed out of a diagnosis set.

Definition 4.1 (Program Diagnosis) *Let Π be an $\mathcal{EXP}$ program, q a query, $\mathcal{G} = (V, Ar)$ the associated evaluation graph, E the expected value, and $\Delta \subseteq V$ a diagnosis. A* program diagnosis Δ_Π *is given by* $\Delta_\Pi = \{expr(v) | v \in \Delta\}$

Our experiments using this simple system description have shown that it approach can be effectively used for diagnosing non-recursive functions. Diagnosing recursive functions leads to the observation that all diagnosis components may cause the fault if only one expected value is given. If several measurement selections have been done, the approach can be more effective even for recursive functions. However, the approach of course does not explain the fault, it merely points out the location of the problem. Its main advantage is that it is fast enough for practical use even for larger examples.

5 Using Expression Replacements for Diagnosis

The main disadvantage of the approach presented in the previous section is that is of very limited capability with regard to judging the quality of diagnoses. To achieve better discrimination between diagnosis candidates, we search for a replacement that explains the faulty behavior, i.e., if the replacement is applied, the program will compute a correct result. Using a simple search and test algorithm would result in an effectively infinite search space (in principle, the whole program could be replaced by a program computing the same function using a different algorithm), therefore effective pruning methods are required. This section describes a way of selecting possible expression replacements which works well in practice for certain classes of errors. Possible expression replacements include only replacements which may lead to an fault explanation. As an example, assuming the type-correctness of a program (which can be guaranteed by conventional means), it is worse than useless to replace the function `plus` by a boolean `and` without further changes in the program. Other, similar restrictions can be applied for user-defined functions. Consider a function `test` and assume that other functions such as `test2, test3, myAnd, ...` are defined. In this case a programmer would only focus his attention on the functions `test2` and `test3` because of their similar names. Likewise, we assume that variables will often be replaced by variables and constants by constants. Using infinite domains (e.g., $\mathbb{N}$) leads also to the question of how to focus on the optimal set of constants for a replacement, since trying them all is impossible. Here, a number of heuristics are possible, mostly based on syntactical similarity. For example, it is more likely that a programmer will mistype a `5` as a `6` instead of a `9`, or `5000` as `50000` instead of, say, `25325`. We therefore try to restrict the search for constant replacement to the nearest constants.

Expression replacement also introduces a new kind of problem to the diagnosis task. Since the program itself is changed, a terminating program might be changed to a non-terminating one. Theoretically, we would have to prove that a given replacement will not change termination by code inspection. To avoid problems regarding termination proofs we introduce an evaluation limit. Essentially, we stop evaluation if the replaced expression causes a much longer evaluation time (typically specified by the user) than the original (incorrect) one.

Definition 5.1 (Expression Replacement) *Given expressions* e, e_1, *and* e_2, *the* expression replacement *of* $e1$ *by* $e2$ *in* e, *written as* $e\binom{e1}{e2}$, *results in an expression* e' *which is* e *with* $e1$ *replaced by* $e2$.

1. $\forall\, e \in \mathcal{EXP} : e\binom{e1}{e2} = e2$ *if* $e1 \equiv e$
2. $\forall\, x \in (VS \cup A) : x\binom{e1}{e2} = x$ *if* $x \not\equiv e1$
3. $cond(x_1, x_2, x_3)\binom{e1}{e2} =$ $cond(x_1\binom{e1}{e2}, x_2\binom{e1}{e2}, x_3\binom{e1}{e2})$ *if* $e1 \not\equiv cond(x_1, x_2, x_3)$
4. $\forall\, f \in (FB \cup FV) : f(x_1, \ldots, x_n)\binom{e1}{e2} =$ $f(x_1\binom{e1}{e2}, \ldots, x_n\binom{e1}{e2})$ *if* $e1 \not\equiv f(x_1, \ldots, x_n)$

Note that e_1 can only occur once within e, due to the definition of expression identity. We generalize the notion of expression replacements to the application of sets of replacements in the natural manner (see (Stumptner & Wotawa 1996) for more details). We write $\Pi|\Theta$ to denote the application of a set of expression replacements Θ to a program Π.

Finally, we define an ordering on expression replacements according to the number of changes implied by them. The distance $diff$ between two expression is defined as number of different subexpressions, e.g., $diff(x1, x1) = 0$, $diff(x1, f(x1)) = 1$, and so on.

Definition 5.2 (Replacement Diagnosis) *Let* Π *be an* $\mathcal{EXP}$ *program with* δ_Π *its function environment,* q *a query and* E *the expected value of* $eval(q, \iota, \delta_\Pi)$. *A set of expression replacements* Δ *is a diagnosis if* $eval(q, \iota, \delta_{\Pi|\Delta}) = OBS$ *is consistent.*

We incorporate replacement in the consistency-based approach by mapping replacements to fault models. To be exact, we introduce a fault mode $wrongexpr(x)$ which has the replacement expression x as parameter. Proving whether a replacement causes a contradiction is simple. Create a new body with the given replacements and call the evaluator. If the result is not equal to the expected value, a contradiction has been detected.

We use the function $modes : \mathcal{EXP} \mapsto 2^{MODES}$ to denote the set of all fault modes to be considered relative to an expression, where $MODES$ is the set of all fault models. Remember that since this set specifies all possible replacements, it is infinite in general and can be described by $MODES = \{wrongexpr(e) | e \in \mathcal{EXP}\} \cup \{ok\}$. As mentioned above, we are only interested in searching replacements that do not change too much of the original expression. In other words, we assume that the correct expression to be searched for is only a small variant of the wrong one.

6 Computing Expression Replacements

In this section we introduce an algorithm that returns possible replacement expressions for a given expression, up to a certain degree of difference. The restriction that new expressions may not evaluate to the same result as the original one naturally arises because we want to explain misbehavior by showing how to correct it.

The diagnosis algorithm for program debugging needs to be little altered from standard algorithms such as Reiter's (Reiter 1987), with extensions to handle behavior modes for different expressions. This algorithm is sufficient if programs are small and a large number of replacements can be taken into account. However, it is much slower than the evaluation-trace-based algorithm and for programs beyond a few hundred expressions, we examine the usage of an iterative algorithm which uses the evaluation trace to provide preliminary focusing for the replacement-based algorithm.

We use an auxiliary data structure to provide knowledge about available functions, constants, signals, and variables while computing replacements, by associating functions with their types called function type environment $FTEnv$, a set $VARS$, and a function *type* for classifying variables and signals. The function *domain* assigns a range of values to each type.

Definition 6.1 (Function Type Environment) *A function type environment $FTEnv$ is a set of triples (F, T, TS) where F denotes the name of a function, T the type of the resulting value, and TS the sequence of types $< t_1, \ldots, t_n >$ of the arguments $< E_1, \ldots, E_n >$.*

Operations related to function type environments are:

$functT : 2^{FTEnv} \times TYPES \mapsto 2^{FTEnv}$ returns a function environment consisting only of functions having the given result type. $functT(FTEnv, T) = \{(F, T, TS) | (F, T, TS) \in FTEnv\}$

$functTS : 2^{FTEnv} \times 2^{TYPES} \mapsto 2^{FTEnv}$ returns a function environment consisting only of functions having the given argument types. $functT(FTEnv, TS) = \{(F, T, TS) | (F, T, TS) \in FTEnv\}$

Initially we assume that the function type environment has at least the following elements:

1. For $f \in \{plus, minus, mult, div\}$: $(f, \mathbb{N}, < \mathbb{N}, \mathbb{N} >)$
2. For $f \in \{inc, dec\}$: $(f, \mathbb{N}, < \mathbb{N} >)$
3. For $f \in \{greater, ge, equal\}$: $(f, boolean, < \mathbb{N}, \mathbb{N} >)$

Computing replacements is restricted by the maximum difference between the original expression and the expression searched for. It is further assumed that functions which are possible replacements must have the same argument types as the function of the original expression, and that the number of arguments of a function does not change. Note that these restrictions can be removed if so desired without changing the basic approach.

The basic REPLACEMENT algorithm can be found in (Stumptner & Wotawa 1996) and has a time complexity of order $o(|FUNC|^D)$ where $FUNC$ denotes the function type environment and D the maximum expression difference between the original expression and the computed replacement.

Searching for replacements explaining a fault can be improved by using a focus on diagnosis components such as follows: First compute all diagnoses using only the *ok* and *undef* mode. Use the resulting diagnoses as component focus for diagnosing the expression using *ok*, *undef* and *wrongexpr*. In the second step, the replacements are restricted. Functions (operations) can only be replaced by other functions (operations) and variables, signals, and constants can be replaced by expression of a given size.

Algorithm REPLACE_SEM(E,Val,Env,FEnv)

1. Convert the query into an evaluation trace.
2. Let $DIAGS$ be the set of diagnoses computed using the evaluation trace.
3. Let $IMPR$ be the set of diagnoses computed using the query and expression replacements, which are computed as follows:
   - For all $undef(F(E_1, \ldots, E_n)) \in DIAGS$ and all functions (operations) F' with the same input and output types as F, define the mode $wrongexpr(F'(E_1, \ldots, E_n))$ for component $F(E_1, \ldots, E_n)$.
   - For all $undef(X) \in DIAGS$ where X denotes a variable or constant, use the REPLACEMENT algorithm to create possible fault modes.
4. The resulting $wrongexpr(Repl)(Expr)$ diagnoses are returned as modes for the expression $Expr$.

7 Initial Experiments

The REPLACEMENT and REPLACE_SEM algorithms have been implemented in Smalltalk and tested using various examples. We have found that using only replacements without focusing on relevant subexpressions is always more time consuming than using REPLACE_SEM (except in the cases where the focus set returned in the first step is virtually equal to the set of components).

The following table gives the used modes (o ... *ok*, u ... *undef*, e ... *wrongexpr*), the number of subexpressions (components) in the code, the amount of time required and the number of diagnoses computed for several examples. The first and second examples deal with boolean functions, the third is a variant of the recursive multiplication function. In the last step, we always search for replacements with a difference of 1.

	Diagnosis	Modes	Expr	T[s]	Diags
1	Evaluation Trace	o, u	33	3	12
	Expr. Replacements	o, u, e		7	5
	Focused Diag.	"		4	5
2	Evaluation Trace	o, u	29	1	4
	Expr. Replacements	o, u, e		8	5
	Focused Diag.	"		1	5
3	Evaluation Trace	o, u	20	1	18
	Expr. Replacements	o, u, e		17	3
	Focused Diag.	"		17	3

The first example shows the focused method as quite effective. In the second example, which involves only boolean expressions (which can only produce one incorrect value as its domain is $\{true, false\}$), the weak discriminating ability of the Evaluation Trace approach happens to be no problem, and the Replacements approach actually produces more diagnoses due to its ability to distinguish between different explanations (i.e., possible replacements) referring to the same subexpression. The iterated application of the two algorithms produces the same results as the Replacement approach alone with significantly reduced overhead. The last example, involving solely integer-based expressions, shows the other extreme. For this particular query, the evaluation trace approach cannot discriminate between responsible expressions at all and therefore does not help in focusing either. The replacements approach is effective, but predictably slower.

8 Other work

There actually exists a considerable body of work on use of knowledge-based techniques for debugging support. We have already mentioned the work of (Console, Friedrich, & Dupré 1993), which was the initial attempt to directly transport MBD techniques to a software domain. However, the choice of logic programming as the domain implies several differences. First, creating a logical model of the program is significantly facilitated, in effect we are dealing with an almost direct mapping. In trying to move from this relatively limited domain to a more general model of software, the question naturally arises what constructs are required to make other languages amenable to MBD techniques. Modeling becomes a problem in its own right, and analyzing the effects particular faults have on the diagnosis becomes more difficult. What can be carried over is the comparison with the technique of declarative debugging (Shapiro 1983; Naish 1992), which is a major field in its own right in the logic programming community. In particular, the basically iterative approach of the declarative debugging techniques which deal with one fault at a time through continuous user interaction contrasts with the fact that the MBD view can handle multiple faults and can integrate the information provided by a set of test cases (although in our case the computational issues may present a significant barrier).

There has been a number of reasonably successful projects dealing with program debugging in the past, often resulting in a high degree of accuracy and correctness in finding bugs, e.g., (Johnson 1986; Murray 1988). However, all these approaches require extensive additional specifications to be applicable. In the case of (Johnson 1986), the specification consists of a set of building blocks (plans) for programs, a detailed description of how a programmer might misuse them in satisfying the requirements (in effect a classification of program bugs in terms of these building blocks), and finally, a specification of the problem in terms of goals that can be satisfied by combining the required building blocks. The programs handled by this approach can be of significant complexity, but then that applies to the specification as well, which in the examples given has about half the size of the program itself (not counting the library of building blocks and bugs). In (Murray 1988), the requirement is more extreme – it is that of a correct implementation of the program/function under study. When envisioning a typical application environment for an intelligent debugger, it is obvious that a correct implementation will not exist (because the initial implementation will be where the debugger is most needed), and that rarely one will have the time to write a formal goal specification that is half as long as the program itself. Both systems are therefore aimed at use purely in a tutorial environment, where (a) solutions for the student problems are known beforehand, and (b) due to the fact that the same problem will be solved anew many times, writing the specification will bring appropriate returns in all those cases. There is no doubt that these methods will in many cases be able of a finer differentiation and analysis than our approach which is at the moment purely based on analyzing the program itself, but that very property is what makes them useless in a general software development environment.

Finally, our assumption that faults will be of limited size (e.g., a small variation will result in a correct program), restrictive as it may seem, is one that is typically made by researchers involved with intelligent debugging environments. For example, under the name *competent programmer assumption* it is referred to as a basic tenet in (Shapiro 1983) and (Budd *et al.* 1980).

9 Conclusion

In this paper, we have shown how model-based diagnosis can be applied to the debugging of functional programs based on a detailed representation of the semantics of the language. We have presented two different approaches for describing the resulting diagnosis problem, one based on examining a given evaluation trace, and the second on computing repairs of the program by replacing subexpressions.

Overall, the expression replacement approach is more useful than the evaluation trace approach. However finding replacements is a very time consuming task and without additional information not practicable for large programs. In such a case using an evaluation trace for diagnosis is much more tractable. As shown in this paper the evaluation trace approach can be used for computing a focus for the expression replacement approach increasing its performance. However, there is a host of issues that are at the moment unsolved. For example, we are only in the first stages of considering an actual treatment of loops or special handling of recursive functions.

While our experimentation was conducted on an intentionally simple example language, the basic assumptions of this paper can be easily transferred to other functional languages or functional subsets of other languages. This work was originally motivated by our research in a different and more complex context, namely representing detailed language semantics for the diagnosis of large-scale hardware designs in the VHDL language.

References

Bond, G. W., and Pagurek, B. 1994. A Critical Analysis of "Model-Based Diagnosis Meets Error Diagnosis in Logic Programs". Technical Report SCE-94-15, Carleton University, Dept. of Systems and Computer Engineering, Ottawa, Canada.

Bond, G. W. 1994. *Logic Programs for Consistency-Based Diagnosis*. Ph.D. Dissertation, Carleton University, Faculty of Engineering, Ottawa, Canada.

Budd, T.; DeMillo, R.; Lipton, R.; and Sayward, F. 1980. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *Proc. Seventh ACM Symp. on Princ. of Prog. Lang. (POPL)*. ACM.

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

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

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

de Kleer, J.; Mackworth, A. K.; and Reiter, R. 1990. Characterizing diagnoses. In *Proceedings AAAI*, 324–330. Boston: Morgan Kaufmann Publishers, Inc.

Friedrich, G.; Stumptner, M.; and Wotawa, F. 1995. Model-based diagnosis of hardware designs. In *Proceedings on the Sixth International Workshop on Principles of Diagnosis*.

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

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

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

Naish, L. 1992. Declarative diagnosis of missing answers. *New Generation Computing* 10:225–285.

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

Shapiro, E. 1983. *Algorithmic Program Debugging*. Cambridge, Massachusetts: MIT Press.

Stumptner, M., and Wotawa, F. 1996. A model-based approach to software debugging. Technical Report DBAI-MBD-TR-96-05, Technische Universität Wien.