SMART C# DEBUGGER – DEBUGGING C# PROGRAMS USING MODEL BASED DIAGNOSIS

Oana PLATON[1]

Depanarea programelor este una dintre cele mai comune sarcini ale unui programator, astfel încât automatizarea acestui proces este nu numai dorită, dar poate deveni o necesitate. Folosirea diagnosticării bazate pe model pentru crearea unui depanator semi-automat este o idee nouă care uşurează viaţa creatorilor de programe. Această lucrare prezintă o aplicaţie care foloseşte modelarea bazată pe valori a programelor C# pentru descoperirea fără efort a erorilor acestuia.

Debugging programs is one of the most common tasks a programmer must face, so automating this process is not only desired and welcomed, is a necessity. Using model based diagnosis for creating a semi-automatic debugger is an innovative idea that brings together two fields with the ultimate goal of making developers' life easier. This paper describes an application that applies value based modeling of C# programs for finding bugs with minimum developer effort.

Keywords: model based diagnosis, value based diagnosis, semi-automatic debugger.

1. Introduction

Writing programs is an innovative and challenging field, where everything can be tried, everything can be questioned, everything can be improved; there are no limits. In order to create a new piece of software some basics steps have to be followed: requirement research, writing a specification, coding, testing and so on. The most important thing is that the final application behaves according to the specification.

A recent study **Error! Reference source not found.** shows that 60-70% of the developing time is consumed for debugging purposes. Classic ways of debugging assume a lot of attention and experience from the programmer. Most of today's debugging tools (Dbx [2]. , Sdb **Error! Reference source not found.**, VAX Debug **Error! Reference source not found.**) allow users to see the values of variables while the program is running, to set breakpoints at certain lines / instructions and nothing more. The programmer is responsible for choosing the locations inside programs where bugs are looked for. In this way, debugging is more an art than a science, because it asks for intuition and complex reasoning.

[1] Eng., Dept. of Computer Science, University POLITEHNICA of Bucharest, Romania

There are also more advanced debugging tools that offer backward execution (Spyder **Error! Reference source not found.**) or query the behavior of the program (Powell or Linton's OMEGA **Error! Reference source not found.**, LeDoux's YODA **Error! Reference source not found.**).

Fault recognition techniques help a programmer formulate hypotheses about fault identity and fault location. They require either a knowledge base of faults or the program specification as their input. Lint's application **Error! Reference source not found.** is based on knowledge about basic errors and finds common portability problems and certain types of programming errors in C programs. FxCop **Error! Reference source not found.** analyses statements (targets) in managed assemblies (.NET) using rules. When a rule is violated, the programmer receives information about the targets. The messages identify relevant programming and design problems, and, when possible, propose possible ways of repairing the errors.

Program specifications are used in LAURA **Error! Reference source not found.**, ASPECT **Error! Reference source not found.** and PUDSY (Program Understanding and Debugging System) **Error! Reference source not found.**. These applications try to discover the discrepancies between the program and its specifications.

Fault localization techniques help the programmer formulate hypotheses about fault location. Program slicing extracts the statements on which certain variables or instructions depend on, directly or indirectly. The programmer then chooses one of these statements as a location for a possible bug. Static program slicing doesn't depend on the execution of the program and can be determined before runtime. It was proposed by Weiser **Error! Reference source not found.**. Dynamic slicing depends on a test case, and had been introduced by Korel and Laski [14.

To increase performance, heuristics for bug localization can be used in debugging tools (e.g. slicing heuristics from Pan **Error! Reference source not found.** and decision-to-decision heuristics, from Collofello or Cousins **Error! Reference source not found.** use test based knowledge for diagnosing the error faster).

Other debuggers use backtracking for returning the program to a previous state **Error! Reference source not found.**.

The objective of this paper is to describe a semi-automatic debugger for C# programs, using model based diagnosis. As programs grow bigger and more complex, following the variables and the functions in thousands of line of code is now overwhelming. Creating tools that can tell the programmer exactly where to look for a bug has become a necessity.

In the next section, model based diagnosis will be introduced. Afterwards, it will be shown how MBD can be applied for debugging programs (especially C#

programs) and how the process of debugging becomes easier using this method. In the end, this method will be compared with other debugging methods, discussing its viability and usefulness.

2. Model Based Diagnosis (MBD)

Model Based Diagnosis is a new technique used to solve real problems from today's world, in fields like mobile phones networks, electric circuits or spatial exploration. These applications have in common the need of using a *model* that will help the diagnosis when errors can not be found by trying all possible values.

The model describes the structural and the functional behavior of the analyzed system and represents, in most of the cases, a mapping of physical world to a logical description.

There is no common language for representing the model, and a lot of techniques have been proposed to solve this problem. Studies have revealed some indubitable benefits of using model based diagnosis, such as: reusability, understandability, well defined semantics, model separation on abstraction levels, simple ways of presenting meta-knowledge, easy utilization and configuration.

3. Applying Model Based Diagnosis in Programs Debugging

Debugging and model based diagnosis share the goal of finding a diagnose (a solution) for a complex system not functioning properly. In order to be able to unify the two, programs are treated as *open systems*.

In the field of diagnosis, open systems are composed from a large number of components, not necessarily of the same type and with the same behavior. Programs are made of statements that contain variables – lots of them, so it is natural to transform each statement in a corresponding component and each variable in a connection between components. This way, the program can be seen as a structural model. The last part is to add a functional side to this model, using the correct behavior of the statements.

After creating the model, the debugging task is reduced to watching a condition while running the model and to isolating the components suspected of wrong behavior if the condition is not fulfilled.

4. Model Based Diagnosis in Debugging Imperative Programs

Before applying a theory and creating an application that works by the standards of that theory, the first required step is to theorize the theory.

Using the consistency based approach **Error! Reference source not found.**, a *diagnose system* is a tuple (SD, COMP), where SD is a logic theory that

models the behavior of the system (in this case, the C# program) and COMP is a set of components (statements).

The system and a set of observations OBS (in this case, test cases) forms a *diagnosis problem.*

A *diagnose* Δ (possible bug) is a subset of COMP, that respects the assumption that all instructions from Δ are wrong, the rest of statements being all correct, is consistent with SD and OBS. Formally, Δ is a diagnosis if (1) is consistent.

$$SD \cup OBS \cup \{\neg AB(C) \mid C \in COMP \setminus \Delta\} \cup \{AB(C) \mid C \in \Delta\} \quad (1)$$

A component that does not behave as it should (in the case of a program, a statement that contains a bug) is represented by the abnormal predicate AB(C).
The diagnose is based on the fact that, if wrong behavior is noticed, it is logical to assume that there are components with abnormal behavior in order to keep the system consistent with its description and to avoid contradictions. We are looking for finding the minimum number of components whose malfunctioning creates the observed error.

The elements involved in diagnosing a program are evidenced in Diagram 1. As a first stage, the C# program is compiled into an internal representation. The model can be adapted for any language, and the only modification to be made is in the part of language specifications.

Starting with the internal representation and a set of model fragments, a converter translates the program into the logical model used for diagnosis. The model fragments represent a logical description of parts of the model, for example the functional description of the methods. This description must be derived from C# language semantics. A value based model must describe model fragments for all base functions and for all kinds of statements (e.g. for binary operator "+" the result is obtained as *Out = In1 +In2* in case of normal behavior).

Once the model is built (which is done automatically, when the C# program is compiled), the model and the test cases are used by the diagnosis engine for finding bug candidates. The number of these candidates can be minimized by using additional information, such as the values of variables at different locations in program.

The location can be selected with an algorithm of selection measurement (finding the variables that should be investigated at certain lines in the original program). The information about value must be given by user. The other candidates offer a reference to the lines / statements from the program, which facilitates bug discovery.

As shown in Diagram 1, formal models have several forms. The best known are functional dependencies model and value based model. The first one assumes the calculus of variables dependencies within a program. Second is based

on calculating the value of all variables at all times. Both models had been tried in Jade Debugger – a debugger for a subset of Java programs **Error! Reference source not found.**.

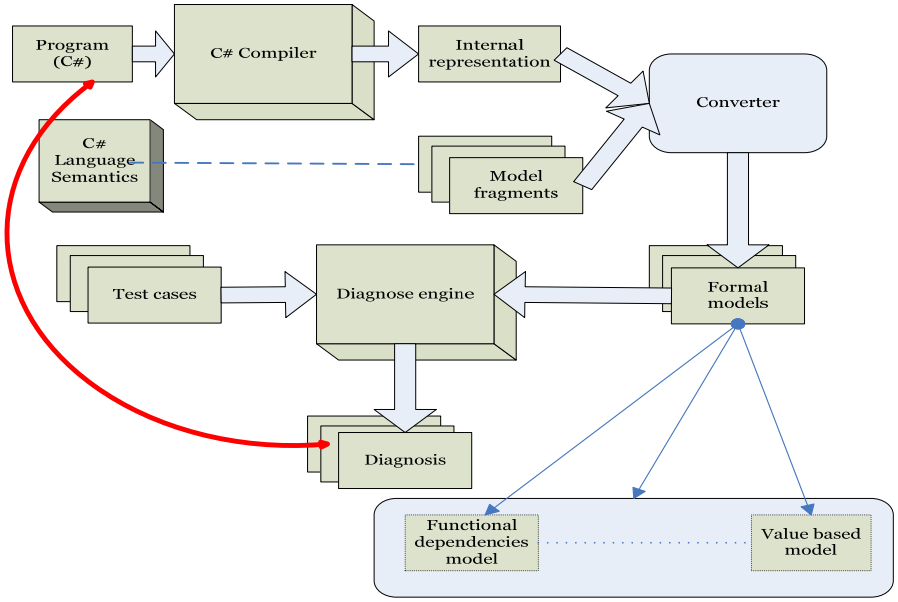


Diagram 1: Steps in debugging a C# program using Model Based Diagnosis

In this paper, value based diagnosis had been chosen, because it gives much more information about the actual location of the bug. The principles of model based diagnosis are illustrated in Diagram 2, which presents the three layers followed in diagnosing a C# program.

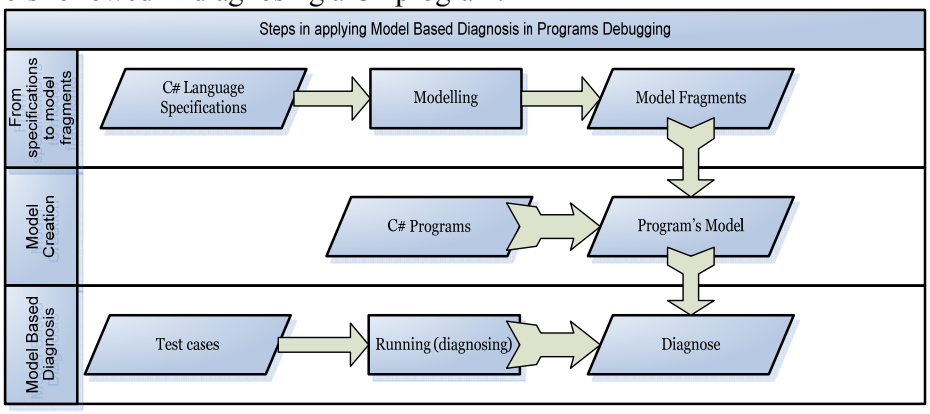


Diagram 2: Model Based Diagnosis layers for program's debugging

5. The Model

To apply MBD in the field of software debugging, the program is automatically transformed in a model, comprised of components and connections between components. The model must have the same behavior as the program, so for same input variables, the output of the model must be identical to the output of the program.

The transformation implies two operations: the statements are converted into components and variables into connections, and represents the structural part of the model. The functional part describes the behavior of the components and does not depend on the program, but only on language semantics. It is added on top of the previous obtained model.

There can be defined an algorithm for constructing the model, based on apparitions of variables. Each variable creates a connection used to connect components. Each time a variable is used in left part of an expression, a new connection is created and used until a new connection appears.

The model is therefore created from components, connected by ports and has inner components as descendants. The C# subset treated at this point is:

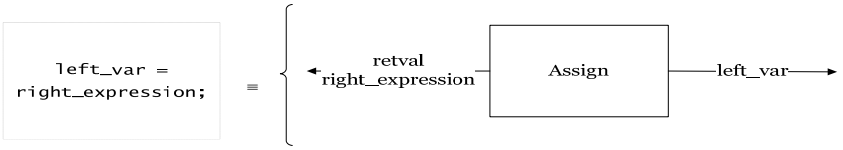
Program := Namespaces ..... [Program]
Namespaces := Classes .....[Namespace]
Class := **class** Id [**:** Id2] {ClassStmnts} .....[Class]
ClassStmnts := ClassStmnt ClassStmnts | ε
ClassStmnt := VariableDecl; | MethodDecl
VariableDecl := Type Id .....[VarDecl]
Type := int | uint | sbyte | byte | string | char | bool | float | double | ushort | short | Id_class
MethodDecl := Type Id (FormalParList) {C#Stmnts } ..... [MethodDecl]
FormalParList := VariableDecl FormalParListRest | ε
FormalParListRest := **,** VariableDecl FormalParListRest | ε
C#Stmnts := C#Stmnt C#Stmnts | ε
C#Stmnt := Assignment | Selection | While | MethodCall | ReturnStmnt
Assignment := Id **=** Expr;..... [Assign]
Selection := **if** (Expr) { C#Stmnts } [**else {** C#Stmnts_*else* }] ..... [Conditional]
While := **while** (Expr) {C#Stmnts} .....[Iteration]
MethodCall := Id (ActualParList) | Id_*Class*.Id (ActualParList)..... [MethodCall]
ActualParList := Expr ActualParListRest | ε
ActualParListRest := **,** Expr ActualParListRest | ε
ReturnStmnt := **return** Expr;..... [Return]
Expr := Id | Constantă | MethodCall | (Expr) | $Expr_1$ Operator $Expr_2$ | new MethodCall.....[Expr]

Once the structural part of the model is created, the functional part (that describes the behavior of each component) must be added in order to debug it.

We use the predicate $AB(C)$ to show that component C has an abnormal behavior, and $\neg AB(C)$ - correct behavior. For each component, we describe the correspondent normal behavior.

Assignment statements [Assign]

Assignments are transformed into components with two ports: an input port, connected at the evaluation of right sided expression, and an output port, created by the variable from the left side. Example:



Instruction 1: Assign

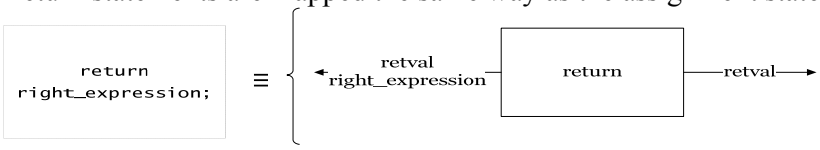
(2) describes the normal behavior of the statement.

$$\neg AB(C) \Rightarrow left\_\text{var}(C) = retval\_right(C) \quad (2)$$

The debugger first evaluates the right side of the expression (the second descendant of the component) and obtains its *#retval* port. For the variable on the left side, it creates a new state and adds the value from *#retval*.

Return statements [Return]

Return statements are mapped the same way as the assignment statements:



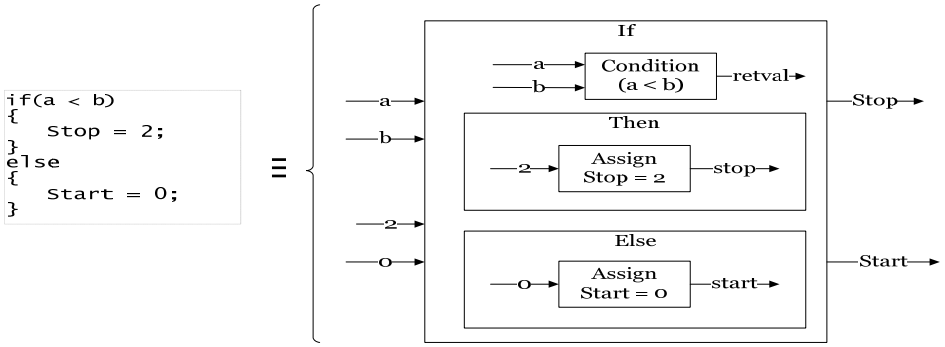
Instruction 2: return

Normal behavior is (2), where *left_var* represents the return value.

Conditionals [Conditional]

Conditionals are mapped to components with variable number of ports. The component contains a subcomponent for the condition, and a subcomponent for each branch. The condition has an auxiliary output port *#retval*, obtained after computing the inner expression. All variables from this expression create input ports for the condition and for the conditional component.

Each variable *x* used as a destination in an assignment statement inside the branches, generates three ports: two in ports connected to the output port regarding *x* from the last statement that changes *x* from then / else branches, and a $\#retval_x$ output port, connected to the future statement that uses *x*. Example:



Instruction 3: If-then-else

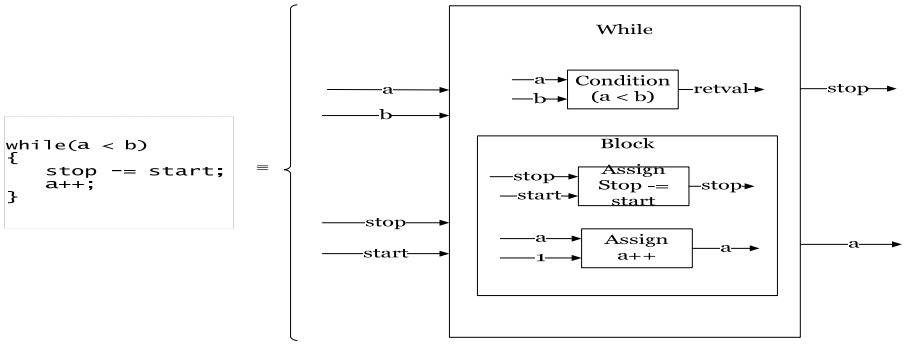
Behavior: For each variable X modified inside any branch we have (3), where *cond(C)* is an in port connected to *#retval(Cond)* (out port for condition), and in port $then_X(C)$ (respective $else_X(C)$) is bound to the out port of the last statement that modifies X in the respective branch.

$$\begin{cases} \neg AB(C) \wedge cond(C) = true \Rightarrow out_X(C) = then_X(C) \\ \neg AB(C) \wedge cond(C) = false \Rightarrow out_X(C) = else_X(C) \end{cases} \quad (3)$$

The debugger evaluates the condition (which is the first descendant) and creates a new state in the *#retval* port of the condition. Depending on the value of #retval (true or false), it executes the corresponding branch.

Iteration statement [Iteration]

The iteration *(while* and *for* loops*)* is mapped to a component with ports associated with the inner variables, containing a subcomponent for the condition and another one for the block. These are converted separately into two separate diagnosis systems, (SD_C, $COMP_C$) for the condition and (SD_B, $COMP_B$) for the block. In the first system, for each variable x used in the condition a connection c_x is created in SD_C. Another connection *#retval* corresponds to the value of the condition. In the second system, f or each in variable x from the block and for each out variable, bo_x a connection bi_x is created in SD_C. Example:



Instruction 4: while

Before actually executing the loop, the debugger checks if it is finite. If the loop contains a break or a return statement in a reachable position, no infinite loop can happen. E.g.:

```
while(true){
        i++;
        break;
}
```

Otherwise, if the variables from the condition never change inside the block, or change but in the wrong direction (the distance between them grew bigger), the debugger prompts the user for a possible infinite loop. E.g.:

```
while(a < b) i++;
while(a < b){
        a--; b++;
}
```

The normal behavior of the component depends on the *#retval* of the condition: if it is true, we use SD_B to discover new values for the variables, values copied in an auxiliary variable v_X^i, where X is the variable and i the current iteration. At the end, the values will be equal with out ports. We use notations VC = variables from condition and VBI (VBO) = in / out variables from the block.

The formal description describes the actual behavior: if the condition is true, execute the block and continue, describing the steps executed from the current iteration to the next one, otherwise end the loop.

$$
\neg AB(C) \Rightarrow \left(\begin{array}{l}
\forall x\, in_X(C) = v_X^0(C) \wedge \quad \forall x\, out_X(C) = v_X^{MAX}(C) \wedge \\
\forall x \in (VC \cup VBI) - VBO \forall I < MAX v_X^I(C) = v_X^{I+1}(C) \wedge \\
\left(\begin{array}{l}
(SD_C(C) \cup \{\neg AB(D) | D \in COMP_C(C)\} \cup \{v_X^I(C) = c_X(C) | X \in VC\}) \\
\Rightarrow cond(C) = true) \\
\rightarrow \forall Y \in VBO \;\; (SD_B(C) \cup \{\neg AB(D) | D \in COMP_B(C)\} \cup \\
\{v_X^I(C) = bi_X(C) | X \in VBI\} \Rightarrow v_Y^{I+1}(C) = bo_Y(C))
\end{array} \right) \wedge \\
\left(\left(\begin{array}{l}
\left(\begin{array}{l}
SD_C(C) \cup \{\neg AB(D) | D \in COMP_C(C)\} \cup \{v_X^I(C) = c_X(C) | X \in VC\}) \\
\Rightarrow cond(C) = false
\end{array} \right) \\
\wedge \neg \exists \;\; J : J < I \wedge ((SD_C(C) \cup \{\neg AB(D) | D \in COMP_C(C)\} \cup \\
\{v_X^J(C) = c_X(C) | X \in VC\} \Rightarrow cond(C) = false \\
\\
\rightarrow MAX = I
\end{array} \right) \right)
\end{array} \right)
$$

The debugger executes the condition, and adds a new state in *#retval* port for this child. If the new state is true, the debugger executes the body of the loop; otherwise, it is terminated.

Method calls [MethodCall]

Method calls are mapped to components with in ports created by global variables used inside the called method or the actual parameters of the method. Out ports are bound to global variables used inside the method, to variables derived from actual parameters of reference types, changed inside the body and to objects created in the called method and accessible from the location where the method is called.

For a method *m*, the component MC is obtained from *m*'s model M by replacing the ports generated by global variables and formal parameters with the actual values and actual parameters. The component has an out port *#retval* that represents the value returned by the method (can be void).

Correct behavior: considering the behavior of the called method Id, B_{Id}, for each rule $(A \Rightarrow B) \in B_{Id}$, we have: $\neg AB(C) \wedge A' \Rightarrow B'$, where A' (and B') are derived from *A* (and *B*) making the replacements described above.

Expressions [Expr]

The expressions are used in statements; the associated component has an out port *#retval* that contains the evaluated result of the expression. Other ports depend on the variables that appear inside the expression. If the expression is:

- **Constant** or **variable** – is added an in port bound to the constant / variable. Correct behavior is described by (4).

$$\neg AB(C) \Rightarrow retval(C) = in(C) \quad (4)$$

- O**perator** – two in ports connected to the *#retval* ports of the inner expressions are added. Correct behavior (5) dependes on the in ports connected to out ports *#retval(Expr)*.

$$\neg AB(C) \Rightarrow in_1(C)\ Operator\ in_2(C) \quad (5)$$

The debugger executes this expression depending on the operators number. It executes the operation given the operator and getting the values of the descendants from their *#retval* ports. The result is added in a new state for the *#retval* port of the expression.

6. Example – creating the model

Let us model the following simple program (StopICondition):

```
public static void Main()
{
     int from = 5;
     int to = 10;
     int start, stop, i;
```

```
      if(from < to)
      {
          start = from;
          stop = to;
      }
      else
      {
         start = to;
         stop = from;
      }
      i = start;
      while(i <= stop)
      {
         i = i+1;
      }
}
```

The model for this program is:

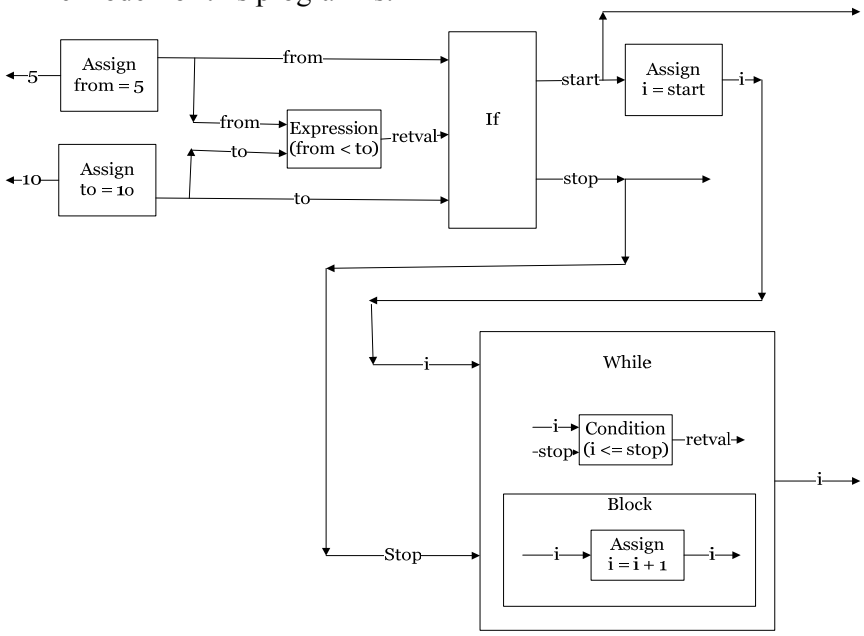


Diagram 3: The model for StopICondition program

7. Debugging the Model

As shown in Diagram 1, the model and the test cases are used by the debugging engine to diagnose a program. Diagnosis is done by following wrong output. To keep consistency in the model, we must assume that there are some components that do not behave correctly. It is desirable to discover the minimum diagnose (minimum number of components whose misbehavior can explain the wrong output).

In our approach, test cases are conditions inserted in components. Conditions can be any expression that evaluates to Boolean values (true/false). The evaluation of the conditions is done by following two principles:

1) Since the evaluation is done at runtime, we know the value of each variable, so we can follow the exact instructions that are executed (which branch in a conditional is executed, how many times a loop is performed and so on).

2) When investigating a component and the error is not found, the investigated program can be reduced to a preliminary state (backward execution).

For StopICondition program we insert the condition `i == stop`. When we run the program we see that the condition fails (`i = 11, stop = 10`).

The last component that modifies one of the variables from the condition is *while*. Here, the error can be in the condition or inside the block. If the error is not in while, we go backwards, and the next analyzed instruction is the assignment `i = start`, then *if* changes `stop`'s value. Assuming that the initializations are correct, we have the candidates: $\{\neg AB(i = start;)\}$, $\{\neg AB(i <= stop;)\}$, $\{\neg AB(while)\}$.

By executing the program backwards, we extract states for variables conforming to the instructions that are rewound. This way, we see that if *while* is executed only five times, not six, the condition is respected. We can apply the principle that tells that most specific information is preferred, so we get as principal candidate the condition of the loop. Indeed, the buggy statement is `i <= stop`, which should be replaced with `i < stop`.

The debugger is also able to find infinite loops in a program, checking whether variables from the condition are modified inside the block in such a way that the distance between them increases.

8. Testing the Debugger

In order to test the application, several syntactically correct C# programs that contained some logical errors have been used. Test examples:

- Programs with infinite loops or possible infinite loops. The debugger can find the problems in proportion of 100%, but gives warnings for programs that contain some finite special loops. E.g.:

```
int i = 100, n = 200;
int x1 = 10, x2 = 20;
while (i < n)
{
        if (x1 < x2) {
            if (1 < i)
                i = i - 1;
            else {
                x1 = 20;
                x2 = 10;
```

```
                }
        }
        else
           i = i + 2;
        n = n + 1;
}
```

- Program that contains many assignment statements, with the following conditions:
  a) Condition that implies all variables that help determine the value of the respective variable;
  b) Conditions that contain only some of the variables. In this case the other variables that influence the evaluation of the condition must be watched.
- Program that contains conditionals with wrong condition, or correct condition and error inside the executing branch. In the last case, the following conditions have been watched:
  a) All statements that contain variables from the list with ports to watch as out ports must be investigated;
  b) None of the statements that do not modify variables to watch should be investigated.
- Program with finite loops, but with some problems, like:
  a) Wrong condition
  b) Errors inside the block.
- More complicated programs, with embedded instructions, with different errors.

9. Evaluating the Performance of the Debugger

Because debugging programs is a NP-C problem (cannot be solved with a deterministic algorithm in polynomial time), our approach uses the oracle's technique (the programmer is asked if at the proposed location is indeed an error).

The time for constructing the model is directly proportional with the number of components that have to be created. The worst time depends on the maximum number of statements (I), the methods called in the converted method (M) and the maximum number of expressions used in the method (E). The number of created components respects relation (6).

$$C \le M(I + E) \tag{6}$$

As seen in Diagram 4, the time needed to create the model is the most significant. This is acceptable, because the model is created once, and then many conditions to investigate can be inserted, and the model can be executed many times.

Programs with thousands of components can be run in several seconds, as Diagram 5 shows. The inserted and investigated conditions do not take more than that, so time needed to diagnose small and intermediate size programs is not big.

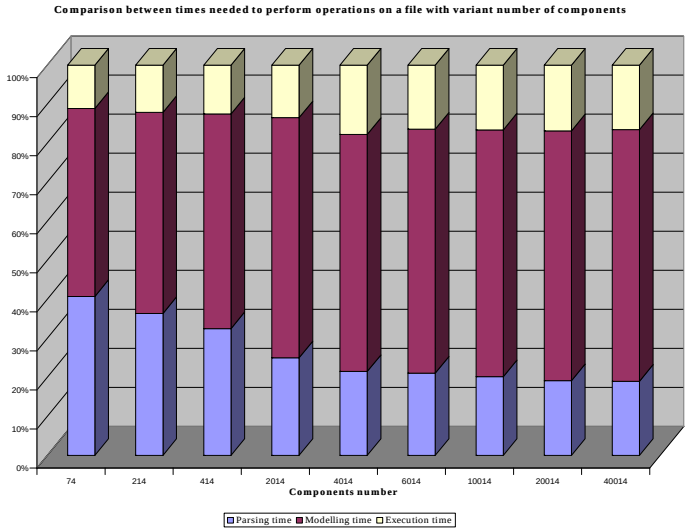


Diagram 4: Comparison between times needed to parse a file, create and execute a model

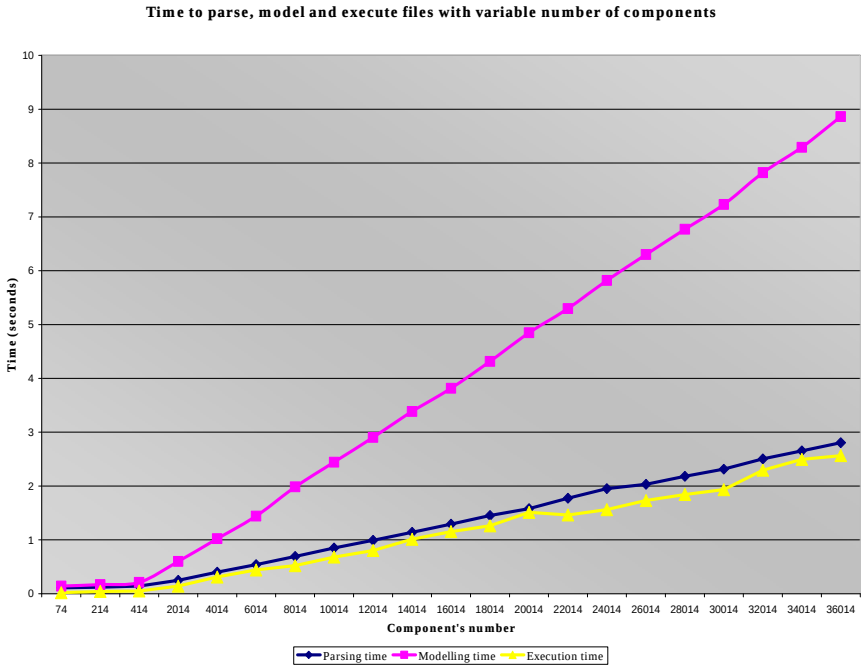


Diagram 5: Times to parse, model and execute a file

10. Conclusions

Automatic debugging represents a difficult problem, because it is NP-Complete. Over the time, partial solutions have been proposed, but none of them solves the problem entirely.

This paper presents a new method, based on MBD. According to the results obtained, the method is viable, the debugger finds infinite loops and then checks conditions and proposes possible bug location to user. If the user doesn't recognize the problem, the program is backward-executed, until the error is found.

The time needed for creating the model is short for small and medium programs (hundred / thousands of components). The debugging time is proportional with the number of components, the number of variables V implied in conditions, and the number of variables that V depends on. The advantage of this approach is that the variables value are known at runtime, so only the statements that are actually executed are investigated (e.g., in if and while).

Locations farther away are investigated just as easily as the closer ones, because to get from a statement to another we simply destroy the intermediate statements that don't change investigated variables.

The debugger has a graphical interface that allows easy manipulation of the debugee, of the conditions inserted in components, and presents information about dependencies and values of the variables at each moment.

Because the application has proved its viability, it is normal to look into the future with new features, like treating all statements in C# language. In this version, only the presented statements are investigated.

Another desired feature is reducing the number of questions presented to the user, by prioritizing the errors (depending on the probability of the error to show up in a location).

A database with possible errors can be created, so that the bugs can be more easily recognized. The debugger can be augmented with reinforcement learning, to understand what mistakes the user usually does (this way, the debugger can be personalized for the programmer). Also, to improve performance, the debugger can use heuristics to minimize the error search space.

Once the debugger had found the error, it should automatically modify the program and continue the execution with the next statement.

A new version will allow the user to insert multiple test cases. The debugger will run the test cases at the same time, finding the bug easier by comparing the conditions for all the data. The debugger can be updated to investigate distributed or parallelized programs.

If the debugger can diagnose MSIL (Microsoft Intermediate Language) code, then it could understand any kind of "managed" language (Common Language Specifications compliant): VB, C#, managed C++, managed Python etc.

In this version, the debugger receives an abstract syntax tree from the C# parser and uses it to create the model. A C# compiler specialized for creating the model had to be written. The performance can increase significantly if the debugger would be integrated with the Microsoft C# compiler.

REFERENCES

[1].http://itpapers.zdnet.com/whitepaper.aspx?kw=software+development&tag=wpr_more&promo=200111&docid=13933

[2]. *Dunlap, Kevin J.* – "Debugging with Dbx", Unix Programmers Manual, Supplementary Document. University of California, Berkeley, CA, April 1986.

[3]. *Katso, H.* - "Sdb: a symbolic debugger", Unix Programmer's Manual, University of California, Berkeley, CA, 1979.

[4]. *Beander, B.* - "VAX DEBUG: An interactive, symbolic, multilingual debugger", SIGPLAN Notices, 173-179, August 1983.

[5]. *Agrawal, Hiralal* – "Towards Automatic Debugging of Computer Programs". PhD thesis, Purdue University, West Lafayette, IN, 1991.

[6]. *Powell, Michael L. and Linton, Mark A.* – "A Database Model of Debugging". SIGPLAN Notices, 67-70, August 1983.

[7]. *LeDoux, C. H.* – "A Knowledge-Based System for Debugging Concurrent Software". PhD thesis, University of California, Los Angeles, December 1985.

[8]. Darwin, *Ian F.* – "Checking C Programs with Lint". O'Reilly and Associates, CA, 1990.

[9]. "FxCop Documentation", Microsoft Corporation, 2004.

[10]. *Adam and Laurent, J. P.* – "LAURA: A system to debug student programs". Artificial Intelligence, 15:75-122, 1980.

[11]. *Jackson, D.* – "Abstract Analysis with Aspect". Proceedings of the 1993 International Symposium on Software Testing and Analysis, 19-27. ACM Press, 1993.

[12]. *Lukey, F. J.* – "Understanding and Debugging Programs". International Journal of Man-Machines Studies, 189-202, February 1980.

[13]. *Weiser, M.* – "Program slicing". IEEE Transactions on Software Engineering, SE-10(4):352–357, July 1984.

[14]. *Korel, Bogdan and Laski, Janusz* – "Dynamic program slicing". Information Processing Letters, 29:155–163, October 1988.

[15]. *Pan, Hsin* – "Software Debugging with Dynamic Instrumentation and Testbased Knowledge". PhD thesis, Purdue University, West Lafayette, IN, 1993.

[16]. *Collofello, James S. and Cousins, Larry* – "Toward automatic software fault localization through decision-to-decision path analysis". Proceedings of AFIP National Computer Conference, 539-544, 1987.

[17]. *Agrawal, Hiralal and Spafford, Eugene H.* – "An execution backtracking approach to program debugging". In Proceedings of the 6th Annual Pacific Northwest Software Quality Conference, 283–299, Portland, Oregon, September 1988.

[18]. *Reiter, R.* - "A theory of diagnosis from first principles", Artificial Intelligence, 57–95, 1987.

[19]. *Mateis, C.; Stumptner, M. and Wotawa, F.* – "Debugging of Java Programs using a Model-Based Approach", Proceedings of the 10th International Workshop on Principles of Diagnosis, Loch Awe, Scotland, 1999.