Dynamic Value-Based Diagnosis System for Assembler Programs

Li Lin[1] and **Yunfei Jiang***[2] and **Zhaofu Fan**[2] and **Chengqi Zhang**[1]

Abstract. Program diagnosis is playing an increasingly important role in computer science. The key steps are general models, slice algorithms and techniques of candidate generation. In this paper, we transferred value-based model and trajectory concept in software diagnosis, which is simple, general and accepted technique applicable to Assembler language program diagnosis system. We have also given a diagnosis system (ADS) and rewritten the PUSH and POP instructions slice strategy which is inconsistent with the definition of conflict set.

***Keywords*:** Value-based diagnosis; Program slicing; Dynamic trajectory; Assembler program

1 INTRODUCTION

Software diagnosis is becoming an important field in software engineering and applications. Its difference from debug is: debug softwares just find syntactic errors, but, software diagnosis means to find semantic errors. This kind of errors often occurs and is very difficult to be detected. Many researchers have finished some related work in this area, such as value-based model [1, 2], trajectory slice [3-5], conflict set [18-19], hitting set [18-19] and PDG [3]. Here, we integrated their methods and presented a system for diagnosing assembler language programs.

A key advantage of value-based model can generate a general model for the production of system descriptions, which can be used to derive diagnosis for differently structured individual systems. This advantage is nowhere more apparent than in the software error diagnosis (or debugging) area. Here, the "value-based" means the computation of slice and diagnosis based the input and output value only.

E.g.1. A symbol function program. (Figure 1)

```
1.      MOV AX, CX   ; CX is unknown
2.      CMP AX, 8000H
3.      JG L1
4.      MOV BX,
5.      JMP END
6.      L1:  MOV BX, 8000H
7.      END: HALT
```

Figure 1. A symbol function program.This program is written in Intel 8086 assembler language. If CX>0 then BX=1, else BX=-1.

Obviously, when CX is less than or equal to zero, BX is evaluated to –1, so in the line 6, "8000H" should be replace by "0FFFFH".

In this paper, an Intel 8086 assembler language diagnosis system was introduced which is implemented in C language under Windows operating system. The foundation of this system is model-based diagnosis principles which is proved by Rieter [1987]. However, the components of hardware diagnosis are instead by statements in software language.

This paper included general model in section 2; slice algorithm in section 3; in section 4, we inferred an algorithm to compute hitting sets; in section 5, we gave an example, we discussed related work in the last section.

2 GENERAL MODEL

The key issue of using model-based diagnosis to debug depends on developing a general model for programs. Such model must be automatically derived from the source code. Moreover, it is independent to the program and suitable to Assembler language. As we know the Assembler programs are different from the other high level language, such as JAVA, C or Pascal. Because it cannot be composed by the three structures: sequence, loop and branch, because of JMP instructions are necessary in Assembler language for improving memory and running efficiency.

Fortunately, there are many researchers have found the strategies to deal with these problems, such as value-based model [C Mateis, 1990], trajectory [B. Korel, 1988], slice [M. Weiser, 1982, 1984, B. Korel 1988], conflict set [J Kleer, 1987, R. Reiter, 1987] and hitting set [R. Reiter, 1987,L. Lin 2003] concepts, now. In this paper, all of these techniques are integrated in the Assembler language program diagnosis system (ADS), and the result is acceptable.

In order to make this paper self-contained, some related definitions and theorems are introduced briefly.

Definition 1 (Diagnosis, [Reiter, 1987]). Let (SD, COMP) be a system and OBS a set of observations. A set $\Delta \subseteq COMP$ is a diagnosis iff

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

is satisfiable.

Here, "AB (·)" means "abnormal" or "work incorrectly", "¬" means "not".

[1] University of Technology, Sydney, Australia, email: {linli, chengqi}@it.uts.edu.au (Li Lin, Chengqi Zhang)

[2] Zhongshan Univeristy, Guangzhou, PR of China, email: {lncsri05, is11}@zsu.edu.cn (Yunfei Jiang, Zhaofu Fan)

* corresponding author.

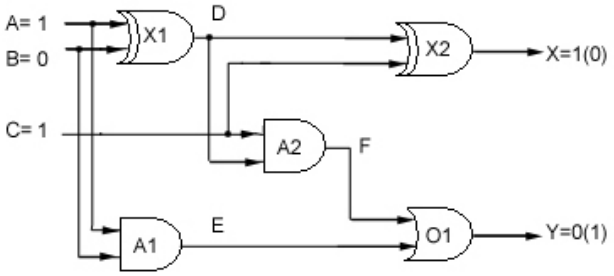

Figure 2. A simple circuit of a full adder.

E.g. 1. (As shown in Figure 2) *COMP*={X1, X2, O1, A1, A2}; "X" stands for XOR-gate, "O" stands for Or-gate and "A" stands for And-gate. *OBS*={A=1, B=0, C=1, X=1, Y=0}.

SD={

…

in1(X1)=A;

in2(X1)=B;

in1(X2)=out(X1);

in2(X2)=C;

…

}

Definition 2 (Conflict set, Reiter, 1987). Let (*SD, COMP, OBS*) be a diagnosis system and *CS* an exact mode assignments. *CS* is a conflict set of (*SD, COMP, OBS*) iff

$$SD \cup OBS \cup \{\neg \mathrm{AB}(C) \mid C \in CS\}$$

is inconsistent. A minimal conflict set as conflict set where no subset is a conflict set.

Definition 3 (Hitting Set, Reiter, 1987). Let C be a collection of sets. A hitting set for C is a set $H \subseteq \bigcup_{S \in C}$ such that $H \cap S \neq \{\}$ for each $S \in C$. A hitting set is said to be minimal if no proper subset of it is itself a hitting set.

Theorem 1 [Reiter, 1987]. A set $\Delta \subseteq COMP$ is a (minimal) diagnosis for (*SD, COMP, OBS*) iff Δ is a (minimal) hitting set of the collection of minimal conflict sets.

So, when we want to get the diagnosis of a system, we just need to compute the minimal conflict sets and hitting sets, which can be computed by Theorem Prover (TP) [R. Reiter, 1987] and Boolean Algorithm (BA) [L Lin, 2003] respectively. For, software debugging, it is difficult to get the standard system description (SD), so, one need to find an efficient concept, slice, to replace system description.

Definition 4 (Programming Language, [Wotawa, 1996]) A programming language L is a tuple (*S, E*) where S denotes a set of logical sentences regarding syntax and E contains logical sentences describing the behaviour.

Definition 5 (Program, [Wotawa, 1996]) Let $L = (S, E)$ be a programming language, A program $\prod$ is an element of $\{x \mid S \alpha\ x\}$. The behaviour of the program is given by E.

Definition 6 (Slice, [Weiser, 1984]). A slice S of a program $\prod$ on a slicing criterion $C = (n,V)$ is any program with the following properties:

(1) S can be obtained from $\prod$ by deleting zero or more statements from $\prod$.

(2) Whenever $\prod$ halts on an input I, S also halts on input I. The resulting values of variables $v \in V$ at position n must be equivalent for both programs.

Similar to minimal conflict set, minimal slice is defined as whose proper subset is not slice. Obviously, the program is a slice itself.

The slice can be divided into two types: static slice and dynamic slice. The static slice is decided by analysing the static program. The dynamic slice depends on actual running environment.

E.g. 2 Static slice. (See Figure 1) The minimal static slice (7, {BX}) is {1, 2, 3, 4, 5, 6, 7}. That is all the statements may be influence BX, however, we know, only one of {4, 5} and {6} can be executed in once running. So, the dynamic slice is {1, 2, 3, 4, 5, 7} when CX is larger than zero and {1, 2, 3, 6, 7} otherwise.

In the next section, we will introduce the slice algorithms which inferred from in [B. Korel, 1988].

3 SLICING ALGORITHMS

The dynamic slicing criterion and the dynamic slice are defined in [B. Korel, 1988] using the executed program path that is named a program trajectory.

E.g. 3. Program trajectory. (Continue E.g.2)

If the input of CX=1, the program trajectory is {1, 2, 3, 4, 5, 7}; if CX=8001, the program trajectory is {1, 2, 3, 6, 7}.

Definition 7 (Dynamic Slicing Criterion, Korel, 1988). Let T be the trajectory of program $\prod$ on input x. A dynamic slicing criterion of program $\prod$ executed on input x is a triple $C = (x, I^q, V)$, where I is an instruction at position q on T and V is a subset of variables in P.

Generally, a dynamic slice is a subset of static slice. Moreover, the result by dynamic may be more exactly and easily to be estimated.

Theorem 2. (Wotawa, 2002) Let $\prod$ be a program of size n, *SD* a logical model of $\prod$ (*SD* = COMPUTE_MODEL($\prod$)), and V a set of variables having a wrong value at position n after executing $\prod$. From V the set of observations is defined as OBS = $\{\mathrm{AB}(v) \mid v \in V\} \cup \{\neg \mathrm{AB}(v) \mid v \in$ variables($\prod$)$\}$. Any slice $(n, \{x\})$ with $x \in V$ is a minimal conflict for (*SD*, {1, . . . , n}, *OBS*), i.e., $SD \cup \{\neg \mathrm{AB}(s) \mid s \in (n, \{x\})\} \cup OBS$ is contradictory.

Proof. (See [Wotawa 2002])

Theorem 2 shows that the (dynamic) slice in software diagnosis is equivalent to the "conflict set" in model-based diagnosis.

As shown in Figure 2, in model-based diagnosis, the diagnosis objects are the components: {A1, A2, X1, X2, O1}, the input, output and connections (OBS and SD) are considered "always normal" even it may be wrong sometimes, so in software program diagnosis, we still discard the abnormal of input and output. It means the expected input and output is always right.

The Assembler language program is difficult to be partitioned into some procedures, because it is unstructured language. So, PDG (Program Dependency Graph, F. Tip, 1995) is more complicated than that of the other languages. However, Assembler language is easier to be compiled and executed. So, the program trajectory can be found quickly and simply. If we get the trajectory, the slice can be found in time complexity O(n) (n is the number of lines).

Program slicing [M Weiser, 1984] is a key technique for debugging. Now, we give an algorithm to compute the dynamic slice of Assembler program.

Algorithm 1. The algorithm of computing dynamic slice.

Step 0. Run the program and find the program trajectory;

Step 1. The program trajectory is obviously a candidate slice (the largest one);

Step 2. Running the program trajectory and get all the relevant output;

Step 3. Let n_1 equal to the first line number;

Step 4. Delete the line n_1 from the slice (program trajectory in the first time) and get a new candidate slice C;

Step 5. Running candidate C again, and compare the current output with the first one;

Step 6. If the result keeps the same, the line n_1 is not in the minimal slice, delete it from the new slice; else, keep it in the new slice set; (value-based)

Step 7. $n_1 = n_1 + 1$;

Step 8. If n_1 is larger than the last line number, stop; else, repeat step 4.This algorithm is according to the definition of minimal slice.

If there are more than one slice, this algorithm just get the "last" one, i.e. the line number is the largest one.

E.g. 4.

```
1. MAIN3  PROC
2.          MOV AX, 3
3.          INC AX
4.          MOV AX, 4
5. MAIN3  ENDP
```

Figure 3. Program MAIN3.

There are two minimal slice for (5, {AX}), {2, 3, 5} and {4, 5}. Here, only the "last" one {4, 5} can be get, but, {2, 3, 5} which useless lose.

In software debugging, the program instructions can be divided into three classes: input/output (OBS), control instructions (SD) and imperative constructions (COMP). So, the software errors can be divided into three types: (1) input/output instructions, (2) control instructions and (3) imperative constructions. In this paper, we just discuss the third (Section 3) type errors; the other will be discussed in the future.

4 ALGORITHM OF COMPUTING HITTING SETS

The method to compute minimal hitting sets is Boolean algebra algorithm, which can process about five-hundred statements and one hundred variables at the same time, it is enough in Assembler program, as shown in Figure 4, see [L Lin, 2003].

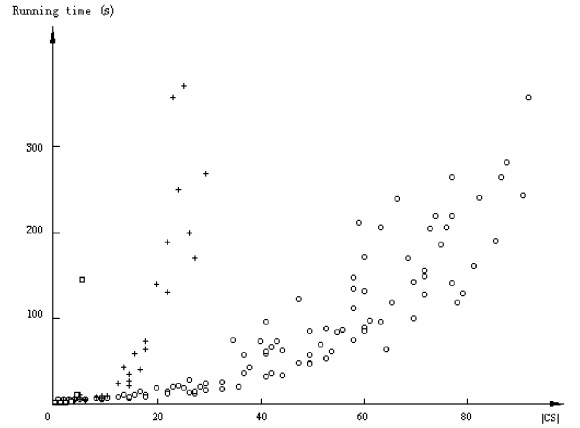

Figure 4. The comparison of HS-tree(□) (the number of |CS| is less than or equal to 7), BHS-tree (+) and Boolean algebra algorithm (O).

5 ASSEMBLER DIAGNOSIS SYSTEM (ADS)

In this section, suppose input/output and control instructions are all right. Such as: "in", "out", "jmp", "jxxx", etc. we debug imperative instructions only, such as: "mov", "add", "sub", "mul", "div", "inc", "dec", "shl", "shr", "and", "or", "not", "cmp", "pop", "push", etc.

5.1. "PUSH" and "POP" instructions

Moreover, this algorithm can get slice including "PUSH" and "POP" instruction.

E.g. 5. PUSH-POP program.

```
1. MAIN4  PROC
2.          PUSH AX
3.          PUSH BX
4.          PUSH CX
5.          POP DX
6.          POP AX
7. MAIN4  ENDP
```

Figure 5. Program MAIN4.

(7, {AX})= {3, 6, 7} according to the minimal slice definition. But, suppose line 5: "POP DX" is missing, (line 5 is not in the slice), but AX value is incorrect either. This contradicts to the conflict definition. AX is wrong, but the conflict {3, 6, 7} all work normally.

To compute the slice by Algorithm 1, we can get the correct result: (7, {AX})={3, 4, 5, 6, 7}. The PDG and revised PDG is shown in Figure 6 and Figure 7.

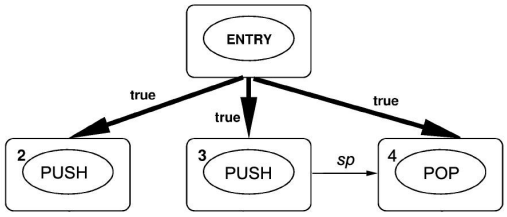

Figure 6. The PDG of MAIN4.

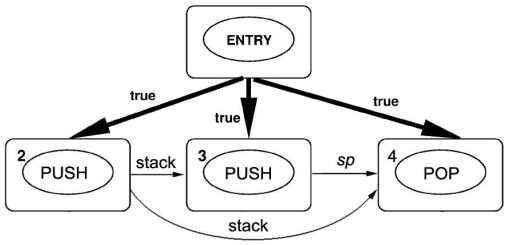

Figure 7. After revised version of PDG.

The slicing algorithm can only get the last dynamic slice if there are more slices. Obviously, only the last one influences the diagnosis result. This method will improve the result, such as static slice or finding all dynamic slices. It can also deal with the problem of "PUSH" and "POP" operators correctly.

We write a program to compute timetable in Figure 8.

E.g. 6. A program to compute timetable.

```
0. START:
1.    MOV CX,0000
2.    MOV AX,1
3.    MOV DX,1
4.L1:
5.    MOV BX,AX
6.    MUL BX,DX
7.    MOV [CX],BX
8.    INC AX
9.    ADD CX, 2 ;(should be "ADD CX, 1")
10.   CMP AX, 0A
11.   JNZ L1
12.   MOV AX,1
13.   INC DX
14.   CMP DX, 0A
15.   JNZ L1
16.END:
```

Figure 8. A program to compute timetable.

According to the definition of slice, ADS removes the statement one by one and tests the result. Through testing, ADS gets the result that only line 1 and line 9 affect the result of CX, so the slice is {CX, 16}={1, 9}.

The ADS user interface is shown in Figure 9.

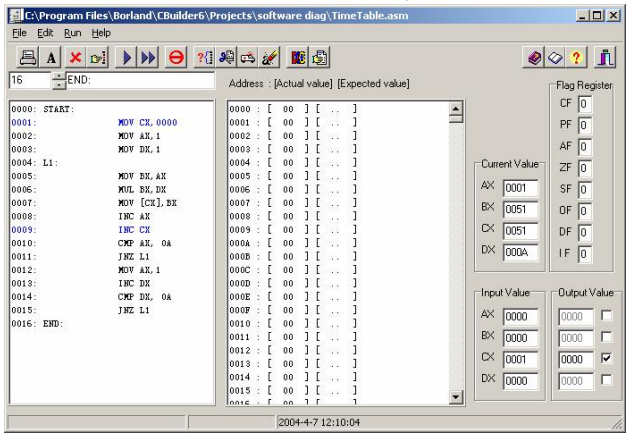

Figure 9. The slicing result (blue lines for register CX).

This program includes 17 lines. The trajectory extends 668 lines.

6 RELATED WORK

Software diagnosis was initially predated by Shapiro several decades ago [E. Shapiro, 1983]. The idea is using programs (in this paper is C language) to find and correct faults in other programs (in this paper is Intel 8086 assembler language). Mateis et al, [Mateis et al, 2000] have finished some research about JAVA programs, they have finished a series papers about this area. Weiser [Weiser, 1984] presented the concept slice that also can be used in diagnosis and debugging, although it is used in program parallel firstly. Korel gave some methods to compute dynamic slice through the executed path -- program trajectory [Borel, 1988] Agrawal improved the algorithm by four approaches, which should improve the efficiency. Some of the proposed techniques use dependencies between variables and statements or knowledge about the program structure specification which is explicitly written by the programmers.

We think that these ideas can be combined with our value-based model (when extended to handle recursive functions) and should provide better discrimination of bug candidates.

7 CONCLUSIONS

In this paper, we have implemented a system can do both slice and diagnosis. The value-based model presented here is practicable and proved by other researchers [Wotawa, 2002, Agrawal, 1990]. The program trajectory extends the original programs to the actual series that can be used explicitly. It can overcome the problem of "JMP". After we get conflict, we can compute hitting sets by Boolean algorithm which costs the less running-time and memory.

All the Assembler instructions are analysed in this paper including PUSH and POP. Moreover, if we want to diagnose a sub-program, the trajectory technique is also a good tool. The algorithms and definitions mentioned in this paper are suitable to Intel 8086 Assembler languages diagnosis, except pseudo instruction. Moreover, it can be transplanted to other type assembler languages easily.

The open problems need to be improved are listed in the follow. Firstly, in general, "program trajectory" makes a program too longer the original program when it contains loop. Secondly, the program diagnosed need be compiled and executed firstly, so, if a language program cannot be compiled or causing more time-consuming, the algorithms mentioned in this paper are not suitable. Thirdly, the correction need be added in the future version. Finally, all the methods are derived from first-principle theories, if expert knowledge is adopted may lead to more detailed and efficiently results.

ACKNOWLEDGEMENTS

Special thanks to Prof. F. Wotawa and Dr. R. Chen from Austria, and S.C. Zhang from UTS Australia, for their helpful comments on earlier drafts of this paper. We also want to thank the referees' advice. The work described in this paper was partially supported by the Natural Science Fund of China (NSFC) under grants 60203015 and 60137039.

REFERENCES

[1] F. Wotawa. (2002). On the relationship between model-based debugging and program slicing. Artificial Intelligence, 135 (1): 125-143.

[2] M. Stumptner, F. Wotawa. (1999). Debugging functions programs. In: Proc. IJCAI-1999, 1074- 1079.

[3] F. Tip (1995). A survey of program slicing techniques. Journal of Program Languages, 3 (3): 121- 189.

[4] M. Weiser (1982). Programmers use slices when debugging. Communication ACM, 25 (7): 446- 452.

[5] M. Weiser (1984). Program slicing. IEEE Transaction on Software Engineering, 10 (4): 352-357.

[6] G.W. Bond, B. Pagurek (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, ON.

[7] L. Burnell, E. Horvitz (1995). Structure and chance: Melding logic and probability for software debugging, Communication ACM, 38 (3): 31–41.

[8] G.W. Bond (1994). Logic programs for consistency-based diagnosis, Ph.D. Thesis, Carleton University, Faculty of Engineering, Ottawa, ON.

[9] L. Burnell, E. Horvitz (1993). A synthesis of logical and probabilistic reasoning for program understanding and debugging, in: Proc. Internet. Conference on Uncertainty in Artificial Intelligence, Washington, DC, 285–291.

[10] S. Danicic, M. Harman, Y. Sivagurunathan (1995). A parallel algorithm for static program slicing, Information Processing Letters. 56: 307–313.

[11] G. Ferrand (1987). Error diagnosis in logic programming, an adaption of E.Y. Shapiro's method, J. Logic Programming, 177–198.

[12] J. Ferrante, K.J. Ottenstein, J.D. Warren (1987). The program dependence graph and its use in optimization, ACM Trans. Programm. Languages System, 9 (3): 319–349.

[13] S. Horwitz, T. Reps, D. Binkley (1988). Interprocedural slicing using dependency graphs, in: Proc. SIGPLAN'88 Conference on Programming Language Design and Implementation, Atlanta, GA, 35–46.

[14] D. Jackson (1995). Aspect: Detecting bugs with abstract dependences, ACM Trans. Software Engineering Methodology, (2): 109–145.

[15] B. Korel, J. Laski (1988). Dynamic program slicing. Information Processing Letters, (29): 155- 163.

[16] B. Korel, J. Rilling (1997). Applications of dynamic slicing in program debugging. in: Proc. Third International Workshop on Automatic Debugging (AADEBUG-97), 43-58.

[17] R.I. Kuper (1989). Dependency-directed localization of software bugs. Technical Report AI-TR 1053, MIT AI Lab, Cambridge, MA.

[18] J. de Kleer, B.C. Williams (1987). Diagnosing multiple faults. Artificial Intelligence, 32 (1): 97-130.

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

[20] L, Lin, Y.F. Jiang (2003). The computation of hitting sets: review and new algorithms. Information Processing Letters, 86 (4): 177-184.

[21] E. Shapiro (1983). Algorithmic program debugging, MIT Press, Cambridge, MA.

[22] F. Wotawa. (1996). Applying model-based diagnosis to software debugging of concurrent and sequential imperative programming languages. Ph.D. Thesis. Technical University Vienna, Austria.

[23] C. Mateis, M. Stumptner, F. Wotawa. (2000). A value-based diagnosis model for Java programs. DX'2000. Morelia, Mexico.

[24] H. Agrawal, J. R. Horgan (1990). Dynamic program slicing. ACM SIGPLAN Notices, 25(6): 246-256.

[25] A.S. Boujarwah, K. Saleh, J. Al-Dallal (2000). Dynamic data flow analysis for Java programs. Information and Software Technology, 42(11): 765-775.