A MILP-based Approach to Path Sensitization of Embedded Software

José C. Costa José C. Monteiro
TU Lisbon, IST / INESC-ID
1000-029 Lisboa, Portugal
Email: {jose.costa, jcm}@inesc-id.pt

***Abstract*—We propose a new methodology based on Mixed Integer Linear Programming (MILP) for determining the input values that will exercise a specified execution path in a program. In order to seamlessly handle variable values, pointers and arrays, and variable aliasing, our method uses memory addresses for data references. This implies a dynamic methodology where all decisions are taken as the program executes. During execution, we gather constraints for the MILP problem, whose solution will directly yield the input values for the desired path. We present results that demonstrate the effectiveness of this approach. This methodology was implemented into a fully functional tool that is capable of handling medium sized real programs specified in the C language. Our work is motivated by the complexity of validating embedded systems and uses a similar approach to an existing HDL functional vector generation. The joint solution of the MILP problems will provide a hardware/software co-validation tool.**

I. Introduction

Embedded systems are used in a growing number of diverse applications. Examples include consumer electronics, automotive systems and telecommunications, among others. This prevalence is due to the fact that embedded systems result from a mix of hardware/software systems. The software part, which runs on a processor, gives the system the flexibility, since it can be easily changed depending on the application. The hardware portion, which executes more specialized functions, is used in time critical subsystems.

Embedded software testing has become more important with the dramatic increase of the size and complexity of the programs. This importance is even more critical since software programs are error prone. Complete path testing, which would give a 100% path coverage, is impractical. Testing only a small set of input values and a small set of paths is the solution. We are left with two problems: decide which set of paths need to be tested while guaranteeing a given confidence level; and determine which inputs need to be applied to the program to activate the selected paths.

In this paper we address the problem of, given a user-specified path, obtaining the inputs that allow a software program to execute this path. Hence, with a judicious choice of a set of paths, the desired coverage level of the code can be achieved. We present a solution to this problem that is applicable to any given high-level language. With the objective of validating our approach, we have implemented a fully functional tool for the C language, which has successfully handled real programs.

By using a similar approach to a hardware method [1] we are establishing a bridge that will enable the integration of the hardware and software methods. A program that solves the joint SAT/MILP program will effectively provide a hardware/ software co-validation tool.

This paper is organized as follows. In Section II, we give an overview of the automated software testing field. In this same section, we also review the vector generation technique from an HDL description. Our method for obtaining the input vectors for path coverage is presented in Section III. Some examples are presented in Section IV. Finally, some conclusions and future work are presented in Section V.

II. Related Work

Generating test patterns for a given program path is a difficult task posing many complex problems. Several methods exist to automatically compute input test data. The existing generation methods are based on symbolic execution [2], on dynamic methods [3], or a combination of both [4], [5].

A. Input Vector Generation

Symbolic execution consists in replacing input parameters by symbolic values and in statically evaluating the statements along a control flow path. A method based on symbolic execution was proposed by Gotlieb et al [2].

Dynamic methods are based on actual executions of programs. A dynamic method based on relaxation techniques was proposed by Gupta et al [6]. In this method test data generation is initiated with an arbitrarily chosen input from a given domain. This input is then iteratively refined to obtain an input on which all the branch predicates on the given path evaluate to the desired outcome.

The DART project [7] developed an approach for generating test cases from symbolic inputs based on both symbolic execution and dynamic methods. Nevertheless DART only handles constraints on integers and does not handle pointer constraints and arrays indexed by variables. The CUTE project [4] extends the DART approach by tracking symbolic pointer constraints. But it still can not handle symbolic pointer offsets and arrays indexed by variables.

EXE [5] also uses symbolic execution and dynamic methods. Due to the solver being used, floating point arithmetic is not possible. Also, all pointer manipulation must be converted into arrays, thus limiting the use of pointer arithmetic (e.g.,

978-3-9810801-5-5/DATE09 © 2009 EDAA

```
1:  function(int i) {
2:   int a[2];
3:
4:   a[1] = 0;
5:   a[i] = 1;
6:   if (a[1] >= 1) {
7:     END OF PATH SPECIFICATION;
8:   }
9: }
```

Fig. 1. Simple example with arrays.

it can not handle a variable that can be reached by double pointer indirection).

None of these methods is able to fully handle pointers and arrays. Furthermore, none of these methods was envisioned to be integrated into a hardware/software test vector generation tool. With the goal of integrating software and hardware test generation methods, we propose a solution based not on software methods but on a hardware testing approach.

B. HDL Functional Vector Generation

Sensitization of a program path is not very different from that of a circuit path. In software, sensitizing a path means that the value at the input of the path will permit the execution of every statement in that path. In hardware, sensitizing a path implies that the value at the input of the path should affect the value at the output of the path.

One algorithm to circuit path sensitization was proposed by Fallah et al [1]. It is a hybrid algorithm for satisfiability checking that seamlessly integrates linear programming (LP) feasibility and satisfiability checking. This integration is necessary due to the correlation between word-level variables and boolean variables.

The algorithm takes as input a circuit described in some high-level language and writes for every module in the circuit a set of input-output relationships. In the case of logical gates, SAT clauses are written. In the case of word-level operators, LP constraints are written. The algorithm then performs a satisfiability search on the SAT clauses where boolean variables are set to $\{0, 1\}$. A search is also done on the LP constraints. The constraints and the SAT clauses are then modified accordingly and a new search begins.

Applying a similar method to software test data generation we will have for each expression in the source program a set of LP constraints.

III. Proposed Methodology

One of the main differences between a hardware high level description language and a software programming language is that in the former the variables can be easily matched with registers or signals. In software, due to the use of pointers, the matching of a variable with its register (in the case of a program the register is a memory position) is not straightforward. Thus, while in hardware we can perform symbolic simulation easily, in software a symbolic simulation will miss some of the aspects of a program (e.g., use of pointers and indirection). Therefore, the method we propose is based on a combination of symbolic (we match the variable name with its memory reference) and dynamic testing (we execute the program to get the memory reference of the variables). Since we actually run the program, all types of data structures can be handled.

In our method, the program is first automatically instrumentalized and compiled, now including routines that track the usage of each variable in the program. The MILP problem is constructed as the program executes, representing all the assignments and conditions that were executed so far. When the program execution reaches the end of the specified path, the solution of the MILP problem defines the values for all of the variables necessary for that path to be executed. Hence, the desired input variables are obtained in a single execution of the given path in the code.

A. Mixed Integer Linear Programming

In our work we use Mixed Integer Linear Programming (MILP) to obtain the input values that allow for the execution of a specified path. That is done by mapping every program variable into a MILP variable. We use Mixed Integer Linear Programming instead of just Linear Programming in order to handle both integers and floating point variables.

While the program is running, each time an assignment statement is executed we add constraints to the MILP problem. When a decision point is reached, we add different constraints to guarantee that the desired branch is taken. At the end of the specified path, we solve the obtained MILP problem. If it is feasible then the solution includes the set of input values that cause the path to be followed. If it is infeasible, then either it is not possible to execute that path, or a backtracking must be made because a wrong value was assumed for some index variable, as discussed next.

B. Backtracking

Backtracking is needed each time we get an infeasible problem to ascertain that that situation was caused by a wrong choice of some array index. Take for instance an array variable. Since each array position maps to a MILP variable, the choice of which MILP variable is being used is made by the value of the index. Thus if we have $a[i]$ and $i = 1$ then the variable that we will be using is $a[1]$. If the use of variable $a[1]$ renders the problem infeasible, this does not mean that there is no input vector for the path chosen. It could be that i must be different from 1 and consequently the wrong variable is being used.

Consider the program in Figure 1. Assume that we want to know the value of i necessary for the program execution to reach line 7. Observing the program code we can readily see that the value of i must be 1. But when the program is running and it reaches line 5 a decision on the value of i has already been made. If the value of i is 1 then the final MILP problem is feasible. Otherwise, we are indexing another position of the array a that is not 1 and when we reach the line 6 we can never make the condition true.

The way we solve this issue is by, each time the problem is MILP infeasible, backtracking to the beginning of the program

```
1:  function(int i) {
2:   int a[2];
3:
4:   AddLPConstraint("# = 0", &a[1]);
5:   a[1] = 0;
6:
7:   AddBacktrack(&a, &i);
8:   AddDependency(&a[i], &i);
9:   SolveLP(&a[i]);
10:   AddLPConstraint("# = 1", &a[i]);
11:   a[i] = 1;
12:
13:   if  (GetBranch()) {
14:     AddLPConstraint("% >= 1", &a[1]);
15:   } else {
16:     AddLPConstraint("% < 1", &a[1]);
17:   }
18:   SolveLP(&a[1]);
19:   if (a[1] >= 1) {
20:     CheckBranch(TRUE);
21:     END OF PATH SPECIFICATION;
22:   } else {
23:     CheckBranch(FALSE);
24:   }
25: }
```

Fig. 2. Instrumenting a program.

and force different values into the array indexes. This is done by adding new MILP constraints that force the array indexes to have values that were not tried before.

C. Implementation

As mentioned before, in order for the program to build MILP problems and solve them during execution, we instrument the original program code. The parser used was `c2c` [8] which is a public-domain software program. `c2c` works by constructing an Abstract Syntax Tree (AST) of a C program. The AST can then be manipulated in several ways such as adding or deleting nodes in it. In our case we add, for each statement in the code, one of a set of functions to the code.

As an illustrative example, consider again the code in Figure 1. When we instrument the program code we get the code in Figure 2. When we run the program we go through all the original statements plus the added ones. Assume in this example that variable `i` is the input variable and that we want to know its value that allows the execution of the path that reaches the END OF PATH SPECIFICATION. The modified program is then compiled and linked with our library of functions.

When we run this program we start by adding a MILP constraint stating that the position one of the array has value zero. This is done by stating the MILP expression (the characters '#' and '%' are used to indicate that they are to be replaced by a variable that is on the left hand side of the statement or by a variable that is on the right hand side of the statement respectively) and by specifying the memory addresses of the statement variables in the order by which they appear in the MILP expression. Then we execute the actual statement. The next statement has a variable that is an array. So, first we add this array `a` with index `i` to the list of arrays and add MILP constraints limiting the index value to the size of the array. After that we state the dependencies (the position of the array depends on `i`) and solve the MILP problem. The variables are updated and we have a value for the array index. Assuming the value of `i` obtained is zero then we are indexing the first position of the array. Then we add another MILP constraint. Next, we reach the `if` condition, so first we have to know which branch we will be taking. If it is the true branch then the constraint `a[1]>=1` is added. Then we will solve the MILP problem if the `if` expression depends on the inputs. In this case the variable `a[1]` depends on `i`. Since constraints `a[1]=0` and `a[1]>=1` make the problem infeasible, we must backtrack. So, we restart the program only this time at line 7 we add a constraint stating that `i` can not be zero. Hence, this time we reach the branch that we wanted. When we solve the last MILP problem we obtain the value of `i` that allow the execution of the specified path.

TABLE I
PROGRAM STATISTICS.

Program	lines	input	decision points	statements
FIBONACCI	24	integer	3	14
LUDCMP	87	array	17	154
DIJKSTRA	141	integer	15	99
HUFFMAN	203	text	17	193
ELEVATOR1	455	bit	40	363
ELEVATOR2	531	bit	80	419

IV. RESULTS

Our method to obtain the test inputs when given a program and an execution path was implemented into a framework.

In Table I we have the statistics of the programs that we used as examples. In it we show the size of the programs in number of lines, the type of input, the number of decision points and the number of statements. DIJKSTRA belongs to MiBench [9], a commercially representative embedded benchmark suite. The implementation of HUFFMAN CODING, LU DECOMPOSITION and FIBONACCI NUMBER are found in Numerical Recipes in C [10].

In order to test several aspects of our framework, the examples above were selected due to their properties: multiplication and division in LUDCMP and HUFFMAN; arrays in DIJKSTRA, HUFFMAN, ELEVATOR and LUDCMP; pointers manipulation and memory allocation in DIJKSTRA and HUFFMAN; multiple functions in DIJKSTRA and HUFFMAN; function recursivity in HUFFMAN; input from the command line in FIBONACCI; input from a file in DIJKSTRA and HUFFMAN.

The examples were submitted to our framework to obtain the instrumentalized code and the obtained code was compiled with our framework. The machine where we run the tests was an Intel(R) Pentium(R) 4 running at 3.2GHz with 1GB of physical memory. The MILP solver we used was `lp_solve` [11] which is based on the Simplex algorithm [12]. In Table II we have the results that we obtained when we tried to obtain an input vector for a certain path and for each of the example programs. The paths were manually

TABLE II
RESULTS OF THE TESTS.

Program	test	branch coverage	max MILP constraints	max MILP variables	problems solved	backtracks	CPU time(s)	memory(kB)
FIBONACCI	F_A	16.7%	1	3	1	0	0.00	1896
	F_B	33.3%	18	10	2	0	0.01	1888
	F_C	66.7%	27	24	4	0	0.02	1896
LUDCMP	L_A	97.0%	2445	2041	37	0	31.10	3652
	L_B	17.6%	129	123	4	0	0.04	2984
DIJKSTRA	D_A	93.3%	186	174	35	0	0.27	2060
	D_B	100%	423	363	72	0	1.00	2200
HUFFMAN	H_A	97.1%	6532	7048	28	0	45.77	7572
	H_B	97.1%	6534	7020	201	25	78.44	23532
	H_C	100%	7362	7969	312	25	310.90	24328
ELEVATOR1	E1_A	75%	1023	1139	7217	299	147.15	16028
ELEVATOR2	E2_A	80%	1867	2289	76033	5158	4359.70	238960

specified with the purpose of obtaining several branch coverages. For each of the tests we have: the coverage obtained; the number of MILP constraints and MILP variables of the most complex MILP problem solved in that test; the number of MILP problems that were solved; and the number of backtrackings necessary to obtain the input values. Finally, for each test we have the CPU time and the memory used.

The results obtained confirm the feasibility of this method. To validate our method we instrumented all programs with a simple function call in each decision point to identify the path being executed. All tests were then validated by using the obtained input values into this modified program, where we computed the branch coverage and asserted that the path being followed was the one specified. In all test cases this was true.

The number of problems solved is directly related to the length of the executed path. It may be greater than the number of decision points of the program if it iterates through some loop. It may be less, if not all of the decision points of the program need to be exercised. The number of problems solved is also related to the number of backtracks.

Some of the tests did not required backtracks to obtain the input vector because they did not have arrays (FIBONACCI) or the array indexes did not depend on the inputs (LUDCMP and DIJKSTRA) or because the first solution found was the feasible one (first test of HUFFMAN). Note also that despite the fact that the programs ELEVATOR1 and ELEVATOR2 have almost the same number of lines in fact the number of decision points is twice as much in ELEVATOR2.

The results for CPU time and memory used are very low even for the larger examples. This gives us confidence that this method is scalable to larger software programs.

V. CONCLUSIONS

We proposed a new methodology based on Mixed Integer Linear Programming (MILP) for determining the input values of a software program that will exercise a specified execution path in the program. Since the MILP constraints are obtained at runtime we can use as identifiers the memory references of the variables. Thus, we can handle every type of variable of a high level programming languge. The test programs we used cover most of the aspects of a high level language and thus show the feasibility of our approach.

In our approach the input vectors are obtained for a specified path. We are currently working on, given a specified coverage value of a coverage metric, obtaining the input vectors that allow for that coverage. With that step completed we can achieve a hardware/software co-validation tool. This can be done either by integrating in our method the hardware functional vector generation or by applying our method to an embedded systems high-level language.

REFERENCES

[1] F. Fallah, S. Devadas, and K. Keutzer, "Functional vector generation for HDL models using linear programming and 3-satisfiability," in *Procs. of DAC*, 1998, pp. 528–533. [Online]. Available: citeseer.nj.nec.com/article/fallah98functional.html

[2] A. Gotlieb, B. Botella, and M. Rueher, "Automatic test data generation using constraint solving techniques," in *International Symposium on Software Testing and Analysis*, 1998.

[3] B. Korel, "A Dynamic Approach of Test Data Generation," in *Conf. on Software Maintenance*, Nov 1990, pp. 311–317.

[4] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," in *Proceedings of the 10th European software engineering conference*. New York, NY, USA: ACM Press, 2005, pp. 263–272.

[5] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: automatically generating inputs of death," in *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*. New York, NY, USA: ACM Press, 2006, pp. 322–335.

[6] N. Gupta, A. P. Mathur, and M.Soffa, "Automated test data generation using an iterative relaxation method," in *Procs. of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, Lake Buena Vista, Florida, United States, November 1998, pp. 231–244.

[7] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM Press, 2005, pp. 213–223.

[8] C2C, "ftp://theory.lcs.mit.edu/pub/c2c."

[9] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *IEEE 4th Annual Workshop on Workload Characterization*, 2001.

[10] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed. Cambridge University Press, 1993.

[11] lpsolve, "http://lpsolve.sourceforge.net."

[12] G. B. Dantzig, "Application of the Simplex Method to a Transportation Problem," *Activity Analysis and Production and Allocation*, pp. 359–373, 1951.