False Path Analysis based on a Hierarchical Control Representation

Apostolos A. Kountouris, Christophe Wolinski

IRISA
Campus Universitaire de Beaulieu
F-35042 Rennes CEDEX, FRANCE
{kountour, wolinski}@irisa.fr

Abstract

False path analysis is an activity with applications in a variety of computer science and engineering domains like for instance high-level synthesis, worst case execution time estimation, software testing etc. In this paper a method to automate false path analysis, based on a control flow graph connected to a hierarchical BDD based control representation, is described. By its ability to reason on predicate expressions involving arithmetic inequalities, this method overcomes certain limitations of previous approaches. Preliminary experimental results confirm its effectiveness.

1. Introduction

False path analysis is an activity with applications in a variety of computer science and engineering domains. It has already been used in high-level synthesis [1], [2], worst case execution time estimation [4], [5], [6] and software testing [8], to mention only a few examples. In path-based scheduling [1] all execution instances (control paths) are considered. The scheduling results can be further ameliorated by using the *false path* analysis technique described in [2]. Identification and elimination of false paths results in a smaller number of control states, increased resource sharing (e.g. operators, registers) and less control logic. As it was shown in [3] such an approach yields a limited number of false paths. Nevertheless, the usefulness of false path analysis for optimization purposes becomes evident. Another area where *false path analysis* can be used is in estimation of *Worst Case Execution Time* (WCET) bounds used in *scheduling* and *validation* of real-time systems. Identification and elimination of false paths is necessary in order to limit the pessimism and make the estimated WCET bounds tighter. In [4], [5], [6] lack of tools led in adapting a manual and thus unsafe approach.

In this paper we propose a method for the identification of false paths. It is based on a special control flow-graph representation connected to a hierarchical BDD based, program control representation which has some similarities to *guard* control representation of[7]. This representation is called the CCFG and is described in section 2. In section 3 false path identification is described and a graph based reasoning on branching conditions that depend on simple arithmetic relations is elaborated to extend even more the false path identification capabilities. A similar technique based on a set of axioms and theorems was proposed in [3]. To handle more complex cases represented by systems of linear arithmetic inequalities, existing formal verification/compilation techniques, i.e. [19], can be adopted. The effectiveness of the false path identification process is demonstrated in a greedy algorithm that enumerates all feasible paths. This algorithm is described in section 4. Finally, our conclusion is drawn.

2. The Conditional Control Flow Graph

Control Flow-Graphs (CFG) have been used to represent the program control flow. Vertices correspond to control *branch*/*join* points in the source code and edges to straight-line code segments. The CCFG is a CFG variant having edges labeled by clocks. A clock indicates the condition under which the control flow passes through an edge. Clocks are organized in hierarchy which will be explained later on.

The CCFG consists of a set of vertices (nodes), a set of edges and a control hierarchy: $CCFG = \{V, E, CH\}$. A vertex v has a set of incoming (*ins*(v)) and a set of outgoing (*outs*(v)) edges. The cardinalities of these sets are *indegree*(v) and *outdegree*(v). There are two distinguished vertices *start, end* which correspond to a program's unique entry and exit points respectively; *indegree*(*start*) = 0, *outdegree*(*end*) = 0. An edge e has a source (*src*(e)) and a destination (*dst*(e)) vertex. *clock(e)* denotes the condition under which the execution flow passes through e; $clock(e) \in CH$.

Suppose that to go from a vertex v_1 to a vertex v_n there are more than one possible paths. Let, $P_{v_1 \rightarrow v_n}$ be the set of paths for going from v_1 to v_n. Each such path is an ordered set of edges indexed in $[1, k]$, $\{e_1, ..., e_k\}$ with:

$src(e_1) = v_1$, $dst(e_k) = v_n$, $dst(e_i) = src(e_{i+1})$. Now for each path $p \in P_{v_1 \to v_n}$ let H_p the set of edge clocks of p indexed in $[1, k]$. We define the clock h_p of path p as:

$$\forall h_i \in H_p, \left(h_p = \prod_{i=1}^{k-1} h_i \right)$$

2.1. The control hierarchy

To construct the hierarchical control representation a dataflow internal representation of the source code is necessary. This representation is called the *Hierarchical Conditional Dependency Graph* (HCDG) and is a special kind of *directed* graph that represents data and control dependencies from a *data-flow* perspective. Historically, it has been developed as internal representation of systems described in the SIGNAL language [9], used for the specification of reactive, real-time systems. The HCDG consists of the *Conditional Dependency Graph* (CDG) and the *Clock Hierarchy* (CH). The CH will be connected to the CCFG by linking each conditional predicate in the input description to the corresponding clock in the HCDG.

Both HCDG nodes and edges are labeled by clocks. Each node corresponds to an operation that assigns a value to a variable. Clocks are a special type of nodes and correspond to boolean conditions that *guard* the execution of operations and the assignment of values to variables, and are similar to the guard conditions of [7]. Edges correspond to precedence constraints (dependencies) on the nodes. The are mainly two types of dependencies: *data* and *control*. The former indicate the values of the nodes that need to be computed before the value of another node is computed, and the latter which conditions need to evaluate to *true*, before a data value is computed.

In a discrete time model, where time is considered as an infinite sequence of logical instants, a *clock* is the set of logical instants that the boolean condition defining it, evaluates to *true*. The theoretical foundations of the HCDG consider clocks as sets and clock formulas as application of set operations on these sets. In [11] it is shown how an equivalent representation of clock formulas as boolean functions can be obtained and vice-versa. Clocks are equivalence classes of the HCDG nodes. Nodes labeled by the same clock carry a value at the same logical instants. In addition an inclusion relation can be defined on clocks. A clock is included in another clock if it is a subset of it. Based, on this inclusion relation it can be said that a clock is more or less frequent than another clock.

The clock nodes of a HCDG are organized in a *Clock Hierarchy* (CH) which is a hierarchical tree-like, representation of the design's control. Such a data structure represents the *inclusion relation* between clocks. In [10] it is shown that this information is very important in order to *triangularize* a larger number of systems of clock equations than it would be possible by using a rewriting system based only on the axioms of boolean algebra. In [11], the clock hierarchy is implemented as a hierarchy of BDD's (*Binary Decision Diagrams* introduced in [12]). BDD's are *canonical* representations of boolean functions, on which boolean operations can be performed efficiently in terms of time and space. Using BDD's two things can be easily achieved; first, *equivalence* between clock formulas can be easily established resulting in a minimal internal representation by avoiding redundancy, and second, during the hierarchization process, it is easy to find the *maximum* depth in the tree that a clock node can be inserted, by means of a factorization process on the canonical representation. This yields an optimally refined inclusion hierarchy.

Inclusion relation. Lets denote by h_i the boolean function corresponding to clock H_i. This boolean representation evaluates to *true* whenever H_i is present otherwise to *false*. The inclusion relation represented by the tree like structure of the clock hierarchy simply states that:

$$\forall (H_i \in descendants(H_i)) \Rightarrow H_j \subseteq H_i$$

Using the boolean definitions the inclusion relation between two clocks will be denoted as:

$$H_2 \subseteq H_1 \equiv h_2 \leq h_1$$

In addition, inclusion can be extended to the following cases:

$$H_k = H_i \cup H_j \Rightarrow H_i \subseteq H_k, H_j \subseteq H_k$$

$$H_k = H_i \cap H_j \Rightarrow H_k \subseteq H_i, H_k \subseteq H_j$$

3. False Path Analysis

To identify false paths in the CCFG we rely on clock information and an algorithm that makes appropriate use of it. In terms of the CCFG definitions for path p:

$$p \in P_{start \to end}$$

A path is *feasible* if the *boolean product* of its edge clocks is not *false*. Otherwise, it is a false path. A CCFG path p is false if:

$$\left(h_p = \prod_{i=1}^{k-1} h_i \right) = 0$$

This is graphically shown in the example 1 of figure 1. h_1, h_2 correspond to the conditions that the first branch is taken or not taken (predicate "a && b" *true* or *false* resp.). Similarly, the possible outcomes of the second branch are represented by h_3 and h_4. Labeling the CCFG edges with these clocks, 2 out of the 4 possible paths in the CCFG can be identified as false.

As far as clock information is concerned, in order to be able to discover a larger number of false paths it is necessary to develop or adopt some form of reasoning about the intersections of clocks that are defined by means of *arithmetic relations* on data values. Such clocks may occur very frequently in practice, especially in the descriptions of control dominated systems. The example 2 in figure 1 graphically depicts such a situation. The CCFG is the same as for example 1. The only difference is that the conditional predicates are no longer pure boolean expressions but contain arithmetic inequalities. In such cases simply calculating the boolean products is no longer adequate for the identification of the two false paths.

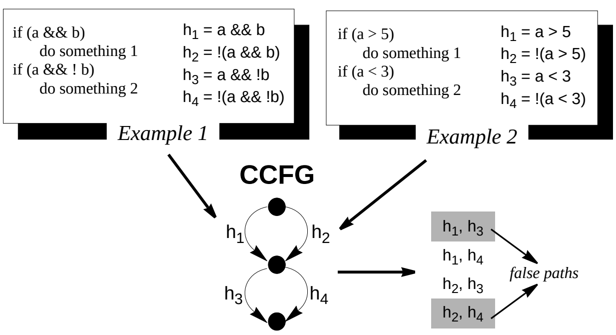

Figure 1. False path analysis examples

3.1. A constrained graph based reasoning

In this section a constrained reasoning for simple relations will be described. This method has some similarities to the technique elaborated in [3].

Lets define an arithmetic *relation* as: $R = A \rho B$ with $\rho \in \{>, <, \geq, \leq, =, \neq\}$ where A, B can be arithmetic expressions and R is a boolean variable being *true* when the relation is true and *false* otherwise. A special form of relation is the *strict relation* where $\rho \in \{>, <, =\}$. We define a *relation set* as a set of relations and similarly a *strict relation set* as a set of strict relations. The intersection of clocks defined by such relations is the conjunction of all the relations in the relation set. The most general case of relations is the one that A and B represent arbitrary arithmetic expressions. Here we shall concentrate on *simple relations* for which A and B represent single variables or constants.

Considering relation sets containing only *simple strict relations*, their conjunction can be represented as a directed graph (the *relation graph*) where a node corresponds to either a variable or a constant and an edge *(source, destination)* to a "*greater than*" relation. Constructing a relation graph the equality operator between two values results in merging the corresponding nodes into a new node. If the relation graph contains a cycle then the conjunction is always *false*. To demonstrate the process lets consider the following simple example. Let A_1, A_2 be variables having the same clock, and L_0, L_1 and L_2 booleans defined by the following relations on A_1 and A_2:

$$L_0 = A_1 < A_2 \qquad L_1 = A_1 > 5 \qquad L_2 = A_2 < 5$$

The corresponding relation graph shown in figure 2, contains a cycle meaning that the conjunction of the relations will always evaluate to *false* for every pair of values for the variables A_1, A_2.

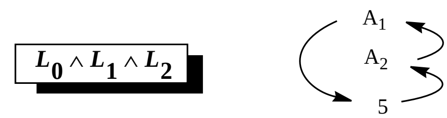

Figure 2. A cycle in the relation graph

This reasoning can be extended to *non-strict* simple relations by transforming the relation set into a set of *strict* relation sets and considering the *disjunction* of the *conjunctions* represented by each strict relation set. For instance, relation $A \geq B$ is transformed into the disjunction of strict relations $(A > B) \vee (A = B)$. Transforming non-strict relations in a relation set this way, we obtain a set of strict relation sets. For each such set we can construct the corresponding relation graph. The only difference is that in order to evaluate the conjunction of the initial relations we have to look for a cycle in *every* relation graph. If all the relation graphs contain a cycle then the conjunction evaluates to *false* for every value of the free variables. The opposite case that there is at least a relation graph with no cycle, means that there exist values for the free variables for which the conjunction of the relations does not evaluate to false.

With this type of graph based reasoning we can effectively decide whether clocks defined by simple relations are mutually exclusive or not. To check for a cycle in a graph we may use well known algorithms like for instance, the DFS (*Depth-First-Search*) algorithm [13].

3.2. Treating complex linear inequalities

Extended reasoning on clocks defined by *arbitrary* arithmetic relations (inequalities), can be achieved using existing compilation techniques like the *Omega test* [15]. The *Omega test* [15], is a system, based on the *Fourier-Motzkin Elimination* [14], for manipulating sets of *affine* constraints over integer variables (presburger formulas). It was initially conceived for dependence testing, and it was designed as a decision test for the existence of integer solutions to affine constraints. The *Fourier-Motzkin Elimination* consists in successively eliminating a variable from a system of linear constraints and corresponds to the successive projection of a *n*-dimensional object to its *n-1* dimensional shadow. If at the end the shadow is empty then the original system has no real solutions.

In addition, techniques and tools from the formal verification domain can also be used. The SVC [19] has the capability of handling propositional expressions containing arithmetic inequalities. Tailoring these methods to our needs to reason on the exclusiveness of clocks defined by linear inequalities, false path analysis results can be significantly improved.

4. A *Greedy* Algorithm for Path Generation

In this section we outline the *findAllPaths* algorithm that progressively constructs the set of all possible paths from *start* to *end* in the CCFG. During the graph traversal we construct for each node a *path list* that contains the paths leading to the node. Each path in this list is represented by an *ordered list* that contains the edges making up the path, and a *clock list* containing the clocks of the path edges. The *path clock* is the *boolean product* of all the clocks in the *clock list*. This algorithm has the characteristic that as false paths are identified, they are eliminated from the subsequent iterations thus decreasing the number of paths that need to be checked for falsehood in subsequent steps. The pathlist of the end node contains all the paths in the CCFG that are not false.

4.1. An example

To demonstrate the ideas presented in the previous sections, we use an example, taken from [16]. The description and the CFG of the example are given in figure 3. To label the edges of the CFG with clocks and produce the CCFG, the specification was parsed into the HCDG shown in figure 4.

```
process jian(a, b, c, d, e, f, g, x, y)
in port a[8], b[8], c[8], d[8], e[8], f[8], g[8];
in port x, y;
out port u[8], v[8];
{
    static T1;
    static T2[8], T3[8], T4[8], T5[8];
    T1 = (a +_1 b) < c;
    T2 = d +_2 e;
    T3 = c +_3 1;
    if (y) {
        if (T1) u = T3 +_4 d; /*u1 */
        else if (!x) u = T2 +_5 d; /*u2 */
        if (!T1 && x) v = T2 +_6 e;
    } else {
        T4 = T3 +_7 e;
        T5 = T4 +_8 f;
        u = T5 +_9 g; /*u3 */
    }
}
```

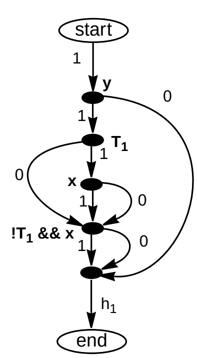

Figure 3. Description and control-flow graph

For readability reasons clock nodes and most of the control dependencies (dashed arrows) have been omitted. Clocks H_4 to H_{10} depend on the boolean result of the comparison node and thus nodes tagged by these clocks transitively depend on it as well. In the HCDG every operator is labeled by a clock. As it can be seen the output variable *u* has multiple definitions (*u1*, *u2*, *u3*) under mutually exclusive conditions and so the single assignment principle is not violated. The resulting clock hierarchy is shown in figure 4, and in table 1 the boolean definitions of each clock are given.

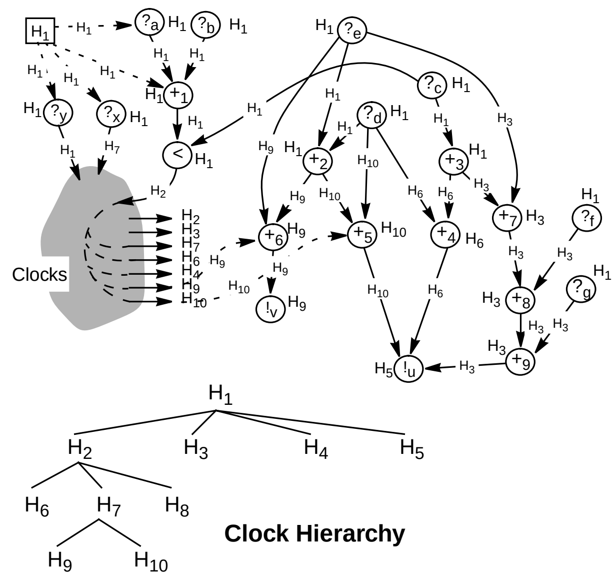

Figure 4. HCDG for the example

Clock	Boolean Definition	Clock	Boolean Definition
H_1	1	H_6	$y \cdot T_1$
H_2	y	H_7	$y \cdot \overline{T_1}$
H_3	$\bar{y}$	H_8	$y \cdot T_1 + y \cdot \overline{T_1} \cdot \bar{x}$
H_4	$\bar{y} + y \cdot T_1$	H_9	$y \cdot \overline{T_1} \cdot x$
H_5	$\bar{y} + y \cdot T_1 + y \cdot \overline{T_1} \cdot \bar{x}$	H_{10}	$y \cdot \overline{T_1} \cdot \bar{x}$

Table 1. Clock definitions

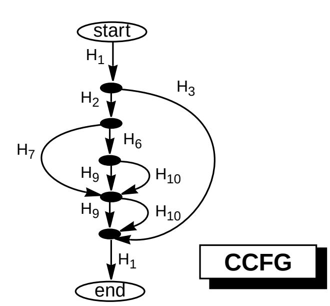

Figure 5. The CCFG of the example

Once the clock hierarchy is constructed for each conditional predicate in the initial description the corresponding clocks for its true/false outcome are found and are attached to the appropriate CFG edges. In this way the CFG is linked to the clock hierarchy yielding the CCFG representation shown in figure 5. Applying the path generation algorithm out of 7 paths only 3 are feasible. All the false paths are successfully identified and eliminated.

5. Experimental Results

Some experimental results are given in table 2 for a set of benchmark programs. Benchmark "*fancy*" taken from the pending section of HLS'92 benchmark suite, has all its conditional predicates defined by arithmetic inequalities and it demonstrates the efficiency of our inequality reasoning method.

Bench.	source	Paths (total)	Feasible Paths	False Paths (%)
jian	[16]	7	3	57
fancy	[17]	162	31	81
ex2	[2]	8	4	50
seat_belt	[18]	12284	15	99.87

Table 2. Benchmark results

The "*seat_belt*" benchmark, first described in [18], contains 17 conditional branches at various nesting levels, including predicates involving arithmetic relations, the *findAllPaths* algorithm found 15 feasible paths out of 12284 possible ones. The *peak* number of paths during the execution of the algorithm did not exceed 55.

6. Conclusion

In this paper a method for false path analysis was presented. It is based on a special control flow graph representation, the CCFG, which is a classical CFG linked to a hierarchical control representation in BDD trees. This method permits to effectively identify infeasible (false) paths and could be easily incorporated within existing methods and tools as an extra aid in the false path analysis process. Such a completely automatic approach avoids the possibility of introducing errors by having the user to decide on path infeasibility.

Nevertheless, some limitations do exist. Even though a large number of cases is covered, all false paths cannot be identified at the boolean reasoning level. For instance, contradictory predicates consisting of complex linear or non-linear inequalities cannot be fully analyzed. A graph-theoretic scheme treating simple linear inequalities was described. For more complex linear inequalities existing compilation techniques could be used.

Acknowledgments

The authors would like to thank the anonymous referees for their fruitful suggestions.

References

[1] R. Camposano. Path-based scheduling for synthesis. IEEE Trans. CAD, 10(1): 85-93, 1991.

[2] R.A. Bergamaschi. The Effects of False Paths in High-Level Synthesis. Proc. of the IEEE Int'l Conference on Computer-Aided Design (ICCAD91), pp. 80-83, 1991.

[3] H-P. Juan, V. Chaiyakul, D. D. Gajski. Condition Graphs for High-Quality Behavioral Synthesis. Int'l Conference on CAD, San Jose, CA, 1994.

[4] Chang Yun Park, Allan C. Shaw. Experiments with a Program Timing Tool Based on Source-Level Timing Schema. IEEE Computer, 24(5): 48-57, May 1991.

[5] P. Puschner, Ch. Koza. Calculating the Maximum Execution Time of Real-Time Programs. RR-01-89, Institut fur Technische Informatik, T. U. Wien, Apr. 1989.

[6] Y-T.S. Li, Sh. Malik, A. Wolfe. Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software. Proc. of the IEEE RTSS'95, 298 - 307, Dec. 1995.

[7] I. Radivojevic, F. Brewer. Analysis of Conditional Resource Sharing using a Guard-based Control Representation. Proc. of the Int'l Conference on Computer Design - ICCD'95, 434-439, Oct. 1995.

[8] A. Bertolino, M. Marre. Automatic Generation of Test Path Sets based on the Flow Analysis of Computer Programs. IEEE Transactions on Software Engineering, 20(12): 885-899, Dec. 1994.

[9] P. Le Guernic, M. Le Borgne, T. Gautier, C. Le Maire. Programming Real Time Applications with SIGNAL. Proc. of the IEEE, 79(9): 1321-1336, Sep. 1991.

[10] L. Besnard. Compilation de SIGNAL: horloges, dependances, environment. Ph.D. thesis, Univ. of Rennes 1.

[11] T. P. Amagbegnon. Forme Canonique Arborescente des Horloges de SIGNAL. Ph.D. thesis, Univ. of Rennes 1, Dec. 1995.

[12] R. E. Bryant. "Graph-based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers, C-35(8): 677-691, 1986.

[13] T.H. Cormen, C.E. Leiserson, R.L. Rivest. Introduction to Algorithms. MIT Press, 1990.

[14] G.B. Dantzig, B.C. Eaves. Fourier-Motzkin Elimination and its Dual. Journal of Combinatorial Theory, A(14): 288-297, 1973.

[15] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. Communications of the ACM, 8: 102-114, Aug. 1992,

[16] J. Li, R. K. Gupta. An Algorithm To Determine Mutually Exclusive Operations In Behavioral Descriptions. Proc. DATE'98, 457-463, Feb. 1998.

[17] The High-Level Synthesis '92 Benchmark suit, available from http://www.cbl.ncsu.edu/CBL_Docs/hls92.html.

[18] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli. A Formal Specification Model for Hardware/Software Codesign. Proc. of Int'l Workshop on Hardware-Software Codesign, Oct. 1993.

[19] Clark W. Barrett, David L. Dill, Jeremy R. Levitt. Validity Checking for Combinations of Theories with Equality. Proc. FMCAD'96, Nov. 1996.