Verification of Software Changes with ExpliSAT

Hana Chockler Sitvanit Ruah
IBM Haifa Research Laboratory, Haifa, Israel.

***Abstract*—We describe an algorithm for efficient formal verification of changes in software built on top of a model-checking procedure that traverses the control flow graph explicitly while representing the data symbolically. The main idea of our algorithm is to guide the control flow graph exploration first to the paths that traverse through the changed nodes in the graph. We implemented this idea on top of the concolic model checker ExpliSAT and the experimental results on real programs show a significant improvement in performance compared to re-verification of the whole program, when the change involves a small fraction of paths on the control flow graph.**

I. Introduction

Software for complex systems is usually not written all at once, but is built incrementally, mainly due to maintenance (fixing errors and flaws, hardware changes) and ehnancements (new functionality, improved efficiency, extension, new regulations). Changes are done frequently during the lifetime of most systems and can introduce software errors that were not present in the old version, or expose errors that were present before but did not get exercised, hence changes, if not verified thoroughly, gradually reduce the reliability of a system. This problem is especially acute in systems that have high reliability requirements, leading to a procedure of re-validating the correctness of the whole system after any upgrade or change. Currently, error detection relies mostly on the execution of extensive test suites, which is very time consuming, and thus, expensive; fault localization is mainly manual and driven by experts' knowledge of the system, and fault fixing often introduces new faults that are hard to detect and remove. In addition, the cost of this validation dominates the maintenance costs of the software (it has been estimated that the cost of change control can be between 40% and 70% of the life cycle costs [9]).

Formal verification, and specifically, *model checking* [3], [12], is a method for verifying that a system represented in a formal way is correct with respect to a given formal specification. It is not a surprise that model checking is widely used in verifying hardware designs [4]. In the last years, formal verification of software gains popularity as well, even though formal models for software systems are much more complex than for hardware designs. Roughly speaking, the underlying idea of model checking is checking correctness of a system by exhaustively exploring its reachable state space. The main advantages of model checking are it being fully automatic and fully reliable with respect to the correctness specification, since the process of model checking involves building a logical proof of correctness. In addition, a standard feature of most model checkers is their ability to accompany a negative answer to the correctness query with a counterexample – a trace of a program in which the property (specification) is violated. As in general the state space of programs is prohibitedly large, different techniques are combined with model checking to improve its scalability (see the survey in [6]).

The advantages of model checking make it an attractive technique for verifying large evolving systems. However, even with space-reducing heuristics, most modern software systems are too large to be model checked as a whole. Luckily, when changes are introduced, the previous version of a system is assumed to be correct, and hence only the changes should be verified. While in theory, even a small change can affect all executions of a system, usually this is not the case: a small change usually affects only a small fraction of the executions, and hence verification of a change should be much cheaper than re-verification of the whole system. This strategy, of verifying only the executions affected by a change, is not always easy to implement on top of an existing model checking tool, since many tools consider the whole system at once. Concolic model checkers, i.e, tools such as ExpliSAT [2] that combine explicit traversal of the control flow graph of a program with symbolic representation of its data, are good candidates for implementing this strategy.

Our update-checking algorithm introduces priorities and increases the priorities of nodes in the control flow graph that are located on paths that go through an updated node. In this way, the symbolic executions that are directly affected by an update are examined first, and for sequential programs, if only verification of an update is needed, the process can be stopped after all changed paths are verified. Since, as we mentioned before, the whole system is assumed to be correct, the effort needed to verify a change or an update directly depends on the influence of this update on the whole system. In most cases, small changes have little influence on the overall correctness of the system, and hence the update-checking algorithm will require very little effort.

We applied ExpliSAT with the update-checking algorithm on real examples of programs with several versions that

This work is partially supported by the European Community under the call FP7-ICT-2009-5 – project PINCETTE 257647.

differ from each other only slightly and compared the results with executing ExpliSAT without the update-checking algorithm (that is, re-verification of the whole program regardless of the updates) on the same examples. The experimental results show that adding the update-checking capabilities improves the scalability of ExpliSAT by several orders of magnitude. In fact, in some of the examples the execution without specifically targeting the update did not even finish, while verification of the update took several seconds.

Related work: From the theoretical point of view, the problem of incremental verification, that is, attempting to verify only the change, can be viewed as an instance of *dynamic graph algorithms*. In this setting, a system is represented as a graph and incremental verification checks the influence of small changes in the graph (edge insertion and removal) on the properties that were previously satisfied in this graph, thus reducing the problem of incremental verification to a dynamic graph problem. However, dynamic graph connectivity, one of the main problems in dynamic graph algorithms, and the one that is most relevant to verification, is an open problem, hence this reduction is of limited value in practice [14]. Perhaps the most relevant to our algorithm is the work by Fedyukovich et al. [7], which relies on previously computed function summaries in order to avoid re-verification of the whole system (their technique is implemented in a tool FunFrog). In contrast to this approach, we do not use the results of previous verification at all – we just limit the current verification procedure to the part of the program that was affected by the change. Finally, the problem of incremental verification is also extensively studied in the dynamic analysis, where there is a rich body of work on *change impact analysis* (see, for example, [1], [13]).

II. Preliminaries and Definitions

In this section, we give an overview of the main ExpliSAT algorithm. A basic understanding of the algorithm is needed since the update-checking algorithm is built on top of the basic algorithm.

A. Basic definitions

We start with the definition of the *control flow graph* (CFG), which is an abstract representation of a program. A vertex in a CFG represents a program statement, and there is a designated vertex representing the initial statement of the program. An edge in the CFG represents the ability of the program to change the control location.

Definition 1 (CFG): A control flow graph (*CFG*) is a directed graph $G = \langle V, E, \mu \rangle$ where V is the set of vertices, E is the set of edges, and $\mu \in V$ is the initial vertex.

In order to construct the CFG of an input program, we invoke the *goto-cc* compiler [8], which compiles C and $C++$ programs into *goto-programs*, i.e. control-flow graphs represented as a list of instructions with at most two successors each. Conditional statements have exactly one condition, that is, exactly two successsors, and non-conditional statements have exactly one successor. Edges of the corresponding CFG are annotated with *edge guards* – boolean formulas that represent the condition that must be satisfied if the program changes its control location by traversing the edge. An edge guard is based on the condition of its source vertex.

Definition 2 ($cond(v)$, $guard(e)$): Let $cond(v)$ denote the condition of a conditional statement v. For an edge $e = (v, u)$, $guard(e)$ equals $true$ if v is not a conditional statement, $cond(v)$ if the edge is traversed when the condition is satisfied, and $\neg cond(v)$ otherwise.

Given a goto-program, we build a CFG representing it as follows.

Definition 3 (Explicit State): An *explicit state* s of the program is a pair $\langle v, L \rangle$, where $v \in V$ denotes the current control location (i.e., the vertex in the CFG) and L denotes the valuation of all the variables of the program over their domain. We write $s \models \varphi$ iff the predicate φ evaluates to true if evaluated using L, and $s \not\models \varphi$ otherwise.

Definition 4 (Kripke Structure of a Program): Let the CFG of a program be $\langle V, E, \mu \rangle$. The *Kripke structure* of that program is the triple $\langle S, I, T \rangle$ where S is the set of explicit states, $I = \{\langle v, l \rangle \mid v \in \mu\} \subset S$ is the set of initial states of the program and $T = \{(\langle v_1, l_1 \rangle, \langle v_2, l_2 \rangle) \mid \exists e = (v_1, v_2) \in E \; s.t. \; \langle v_1, l_1 \rangle \models guard(e)\}$ is the set of transitions between the states.

Definition 5 (Execution): An *execution* π of a program is a sequence of explicit states $(s_1, s_2, \ldots, s_n)$ s.t. $s_1 \in I$ and for every $1 \leq i < n. (s_i, s_{i+1}) \in T$. A state s is said to be *reachable* iff there exists an execution π that contains s.

The property we are interested in is *reachability* of state s that violates a given predicate $p(v)$, where v is the control location of s. As an example, if v is a user-specified assertion with condition x, $p(v)$ is x.

Definition 6 (Control Path): A *control path* c of a program is a path through the CFG of the program, i.e., a finite sequence $(v_1, \ldots, v_n)$ of nodes of the CFG where $v_1 = \mu$ and $\forall_{1 \leq i < n}.(v_i, v_{i+1}) \in E$. The set of control paths is denoted by $\mathcal{C}$. If c is a projection of an execution π on V, we call c a *legal* control path.

We denote the projection of an execution π onto the CFG by $cp(\pi)$. The execution π is said to follow the control path $cp(\pi)$. There may be many different executions that follow the same control path. They differ only in the data (i.e., the valuation of the variables).

The *Static Single Assignment (SSA)* [5] form is a representation of a program in which every variable is assigned exactly once. Existing variables in the original representation are split into versions. New variables are distinguished from the original name with a subscript such that every assignment has a unique left hand side. In SSA form, the function returning a non-deterministically chosen input $input()$ is

```
1.   a = 1;
2.   if (a>0) {
3.      a = input();
4.      if (a ≤ 1) {
5.         c = a+2;
6.         assert(c<3);
7.      }
8.      if (a≤1)
9.         a=2;
10.     else
11.        a=1;
12.  }
```

Figure 1. A non-deterministic program

replaced by an indexed variable $input_i$, which denotes the value returned by the i^{th} call to the $input()$ function.

The majority of software verification tools (see, for example, CBMC [11]) translate the whole program into SSA form. In contrast, ExpliSAT does not translate the program into SSA form, but rather stores in each state a guard representing the control path following which this state was reached and the symbolic values of each of the variables in this state. In what follows, however, we will use SSA form in order to demonstrate different control paths of the same program. Since every variable in the SSA form of a control path is assigned exactly once, it can be considered as a set of constraints that must be satisfied in any execution that follows that control path. We denote the conjunction of the SSA constraints of a control path c by $SSA(c)$.

Definition 7 (Path guard): The *path guard* of a control path c is denoted by $cpg(c)$ and is the conjunction of the guards of all edges in c given that c is in SSA form:

$$cpg(v_1, v_2, \ldots, v_n) = \bigwedge_{1 \le i < n} guard(v_i, v_{i+1})$$

B. ExpliSAT algorithm

In ExpliSAT, the algorithm traverses the abstract representation of the program represented by the CFG, rather than the Kripke structure of the program. That is, we explicitly explore only the control paths and use symbolic methods to cover the various executions. Two executions π_1, π_2 are said to be *control equivalent*, denoted $\pi_1 \sim \pi_2$, iff they follow the same control path, i.e., $cp(\pi_1) = cp(\pi_2)$. The equivalence classes that this relation induces aid in decreasing the size of the software model on which an explicit search is performed. Moreover, ExpliSAT maintains *path representatives* for each control graph path ensuring its feasibility[1].

The basic idea of ExpliSAT algorithm is to traverse all control paths of the CFG where each generated state holds the guard of the path from the initial state to the current state. The algorithm maintains a CNF encoding of each control path constructed from the constraints given by the edge and state guards and the assignments to variables. A propositional SAT solver is used for deciding the feasibility of conditional statements branches. For assertions, the SAT solver is used to search for a satisfying assignment to the encoding of the path and the negation of the property associated with the last control location, thus presenting a counterexample.

For example, consider the program in Fig. 1. The guards of the two possible control paths of this program represented in an SSA form are:

$$cpg(Path_1) \iff (a_1 > 0) \wedge (a_2 \le 1)$$
$$cpg(Path_2) \iff (a_1 > 0) \wedge \neg(a_2 \le 1)$$

The SSA constraints of these control paths are given by the following two equivalences:

$$SSA(Path_1) \iff (a_1 = 1) \wedge (a_2 = input_1) \wedge (c_1 = a_2 + 2)$$
$$SSA(Path_2) \iff (a_1 = 1) \wedge (a_2 = input_1) \wedge (a_3 = 1)$$

Let $c = (v_1, \ldots, v_n)$ be a control path. If there exists a satisfying assignment to $\zeta \equiv cpg(c) \wedge SSA(c) \wedge \neg p(v_n)$, then there exists a reachable state s with control location v_n that violates the property $p(v_n)$. By transforming ζ to CNF, we can use a SAT solver to check if there exists an execution that follows c and violates the property. For example, for $Path_1$:

$$\zeta \equiv ((a_1 > 0) \wedge (a_2 \le 1)) \wedge$$
$$\wedge((a_1 = 1) \wedge (a_2 = input_1) \wedge (c_1 = a_2 + 2)) \wedge (\neg(c_1 < 3))$$

III. Update-Checking Algorithm

In this section we describe our algorithm. The main idea is to change the order of the CFG traversal by forcing ExpliSAT to examine the paths that go through an updated node first. Then, ExpliSAT can be terminated or allowed to continue the verification procedure after the whole system is verified. We observe that there can be many paths in the CFG of a program that go through the same updated node. We start from the updated node and raise the priority of nodes in the CFG that reside on all paths from the updated node back to the root of the program, by doing a backward reachability traversal from the updated node to the root. Then, we raise the priority of the nodes accessible from the updated node until the end of the execution by doing a forward reachability traversal from the updated node to the leaves of the CFG.

Clearly, not all paths on the CFG that go through an updated node are feasible. Since our algorithm is a lightweight pre-processing step before the main model-checking procedure, we do not check feasibility of the CFG paths that we encounter. This part is taken care of by executing ExpliSAT with updated priorities.

We added an option of executing ExpliSAT only on traces that pass through the updated node. As we state later, in sequential programs, in order to check the correctness of an update it is enough to check the traces that pass through the updated node; in addition, all found bugs are real bugs

[1]For a detailed explanation of the path representatives the reader is refered to the original ExpliSAT paper [2].

```
function FINDROOTTOUPDATENODES
    Input : Node u, CFG G
    Output: Set nodes_from_root
    Append u to bfs_queue
    nodes_from_root ⟵ u
    while bfs_queue ≠ ∅ do
        q ⟵ FrontOf(bfs_queue )
        for each incoming edge e = (v, q) do
            if e is not repeating then
                if v ∉ nodes_from_root then
                    nodes_from_root ⟵ v
                    bfs_queue ⟵ v
        remove q from bfs_queue
    return nodes_from_root
```

Figure 2. Finding all nodes from the root of the CFG to u

```
function FINDUPDATECONE
    Input : Node u, CFG G
    Output: Set update_cone_nodes
    bfs_queue ⟵ ∅
    Append u to bfs_queue
    Insert u to update_cone_nodes
    while bfs_queue is not empty do
        q ⟵ FrontOf(bfs_queue )
        for each outgoing edge e = (q, v) do
            if e is not a loop-back edge then
                if v ∉ update_cone_nodes then
                    insert v to update_cone_nodes
                    append v to bfs_queue
        remove q from bfs_queue
    return update_cone_nodes
```

Figure 3. Finding all nodes from u to the end of the program

```
procedure FINDUPDATEPATHNODES
    Input: program P, Node u
    Output: Sets nodes_from_root, update_cone_nodes
    Construct the CFG G reachable from the first statement
    nodes_from_root ⟵ FindRootToUpdateNodes(u, G)
    update_cone_nodes ⟵ FindUpdateCone(u, G)
```

Figure 4. Finding all nodes on paths traversing u

in the program. Thus, ExpliSAT with update-checking is an efficient bug-hunting procedure for changes and upgrades, and can also be used to re-certify a previously verified program that was changed.

The update locations are indicated as labels with a predefined label in the input source file which are stored in the node of the generated goto-program.

Let u be a location of an update in the goto-program. Throughout the model checking loop we maintain the following:

- A set *nodes_from_root* containing all nodes on paths from the root of the CFG to u (including u).
- A set *update_cone_nodes* containing all nodes on paths from u to the end of the program.
- *root_to_update_states* - a queue of states to process whose locations are on paths from the root to u.
- *update_cone_states* - a queue of states to process whose locations are on paths from u to the end of the programs.
- *low_priority_states* - a queue of states to process that are not on paths including u, or that are inside loops.

Definition 8: An edge e in the CFG is a *loop-back edge* if it is the backward edge from the end of a loop to the loop head. An edge e in the CFG is *repeating* if it is either a loop-back edge or a recursive function call.

The procedure ModelChecking accepts as input a goto-program P and an update location u in P and traverses the control flow graph of P such that paths containing u are visited before other paths. The procedure first finds all the paths containing u by calling FindUpdatePathNodes. The nodes from the root to u are stored in *nodes_from_root* while nodes that appear on paths from u to the end of P are stored in *update_cone_nodes*.

For a state s, the CFG node corresponding to the control component of s is denoted $s.node$. States whose CFG nodes are in *nodes_from_root* have higher priority than state whose CFG nodes are in *update_cone_nodes*, which in turn have higher priority than states that are not on any update path – these nodes are inserted to *low_priority_states*. Inside the fixed-point loop of line 4 , the next state to process is taken from the state queue with highest priority (line 5).

A new generated state is inserted to the highest priority queue possible (lines 7 to 14). The model checking loop terminates when all state queues are empty.

Claim 9: In sequential programs, only the paths that traverse through the changed node can be affected by a change.

```
1:  procedure MODELCHECKING(program P, Node u)
2:      FindUpdatePathNodes(P, u)
3:      root_to_update_states ⟵ initial state
4:      while some state queue is non-empty do
5:          s ⟵ FrontOf(highest priority state queue)
6:          Generate successors of s
7:          for each successsor t of s do
8:              if t is a new state then
9:                  if t.node ∈ nodes_from_root then
10:                     root_to_update_states ⟵ t
11:                 else if t.node ∈ update_cone_nodes then
12:                     update_cone_states ⟵ t
13:                 else
14:                     low_priority_states ⟵ t
15:         delete s from its queue
```

Figure 5. Model checking program P given an update location u

The correctness of the claim follows immediately from analyzing the dependencies between nodes in the CFG of a sequential program.

IV. Experimental Results

We have implemented our update-checking algorithm on top of ExpliSAT and checked it on several C++ programs. ExpliSAT was previously used internally inside IBM to verify several complex locking protocols in an industrial middleware software. It has the advantage of working with a small state space since it verifies the program one path at a time. On the other hand, when the program is large, verifying it one path after another requires a lot of time. In our experiments, we compared the running time of the original ExpliSAT with the running time of ExpliSAT enhanced with our update-checking algorithm on examples in which a small change was introduced in a large program. ExpliSAT with the update-checking algorithm is configured so that the tool terminates after all paths that pass through a changed node are verified.

Clearly, there is a significant reduction in the running time of ExpliSAT when it is restricted to a subset of control flow paths. Moreover, in several cases, ExpliSAT, when running on the whole program, did not terminate in a reasonable time at all, while with update-checking algorithm it terminated in a number of seconds.

In addition to artificially constructed examples, our experiments included one real-life example – a C++ program supplied by VTT Technical Research Center of Finland [15] (the program is available from the authors on request). The program computes the velocity and acceleration of a robot used in the European ITER project – a new type of a reactor based on energy of nuclear fusion [10]. A new version of the program contained a bug, which was discovered by ExpliSAT in several seconds. In contrast, the full verification was terminated by us after several hours without finding the bug. The bug affected the main path of the program and hence its output.

V. Conclusions and Future Work

In this paper, we presented an algorithm for efficient verification of a single update in a program. Our algorithm does not rely on the results of the previous verification and can be used to focus the verification efforts on the latest change in a large evolving program. In sequential programs, it exhaustively verifies all affected behaviors of the program, and hence is sound and complete if the previous version of the program was verified. The experimental results show, not surprisingly, a significant improvement in scalability of our approach over re-verification of the whole system.

In the future, we will extend our algorithm to handle multiple updates at once. A straightforward approach to handling multiple updates is to apply our algorithm to each update separately, but a more efficient approach is to use the hierarchy between updates in order to verify only the updates which are not in the cone of influence of other updates.

Another direction is to apply our algorithm to concurrent programs. In concurrent programs, it is no longer true that only paths that go directly through a change are affected by it, since a change in a global variable can affect executions of other threads. Hence, a more subtle analysis of dependencies between nodes in a CFG is needed.

References

[1] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *ASE*, pages 2–13, 2004.

[2] S. Barner, C. Eisner, Z. Glazberg, D. Kroening, and I. Rabinovitz. Explisat: Guiding sat-based software verification with explicit states. In *HVC*, LNCS 4383:138–154, 2007.

[3] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, LNCS 131:52–71, 1981.

[4] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[5] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. An efficient method of computing static single assignment form. In *POPL*, pages 25–35. ACM, 1989.

[6] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[7] G. Fedyukovich, O. Sery, and N. Sharygina. Function summaries in software upgrade checking. In *HVC 2011*.

[8] goto-cc compiler. http://www.cprover.org/goto-cc/.

[9] Penny Grubb and Armstrong Takang. *Software Maintenance Concepts and Practices. Second edition*. World Scientific, 2005.

[10] ITER Homepage. http://www.iter.org.

[11] D. Kroening, E. Clarke, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *DAC*, pages 368–371. ACM, 2003.

[12] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proc. 5th International Symp. on Programming*, LNCS 137:337–351, 1981.

[13] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. In *OOPSLA*, pages 432–448, 2004.

[14] G. Swamy, R. K. Brayton, and Vigyan Singhal. Incremental methods for FSM traversal. In *ICCD*, 1995.

[15] VTT Technical Center of Finland. http://www.vtt.fi.