Just Test What You Cannot Verify!*

Mike Czech, Marie-Christine Jakobs, and Heike Wehrheim

University of Paderborn, Germany
mczech@mail.upb.de, {marie.christine.jakobs,wehrheim}@upb.de

Abstract. Today, software verification is an established analysis method which can provide high guarantees for software safety. However, the resources (time and/or memory) for an exhaustive verification are not always available, and analysis then has to resort to other techniques, like testing. Most often, the already achieved *partial* verification results are discarded in this case, and testing has to start from scratch.

In this paper, we propose a method for combining verification and testing in which testing only needs to check the residual fraction of an uncompleted verification. To this end, the partial results of a verification run are used to construct a *residual program* (and residual assertions to be checked on it). The residual program can afterwards be fed into standard testing tools. The proposed technique is sound modulo the soundness of the testing procedure. Experimental results show that this combined usage of verification and testing can significantly reduce the effort for the subsequent testing.

1 Introduction

Today, software verification has reached industrial size programs, with a large number of tools providing an automatic analysis (see e.g. the annual software verification competition [5]). Still, verification tools might fail in analyzing the program at hand. This might have two reasons: (1) the resources necessary for a complete verification are not available, e.g. because an "on-the-fly" analysis is needed, or (2) the property to be verified is beyond reach of the verification technology, e.g. when complex structural properties are involved. In this case, software engineers need to resort to other analysis techniques, for instance the most widely used method of *testing* [4]. In this case, the work done in a prior, but incomplete verification run is usually discarded, and testing is started from scratch, again considering the complete program and set of properties to be analyzed. This seems to be an unnecessary waste of time and effort.

In this paper, we present a method for combining verification and testing in such a way that a testing run following an unfinished verification run *need just test those parts of the program which have not been verified*. Prior combinations of verification and testing most often follow other principles: either one of the techniques is used to generate likely properties which the other technique then

* This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901).

A. Egyed and I. Schaefer (Eds.): FASE 2015, LNCS 9033, pp. 100–114, 2015.
DOI: 10.1007/978-3-662-46675-9_7

has to check (e.g. for likely invariants [29] or potential error locations [6]), or information computed by one technique is used to enhance the other technique (e.g. test data used for abstraction refinement [27]). Computation of *residual programs* (or properties) is employed in none of these approaches. The only other work aiming at a reduction of one parts of the analysis is [12]. They use a value analysis to compute potential errors (so called alarms), and use program slicing [32] on the statements occurring in alarms to reduce the effort of successive testing. The first static analysis is therein executed on the whole program and only the dynamic analysis has a reduced effort. Here, we introduce a true divide-and-conquer type of combining static and dynamic analysis: verification is doing one part (basically as much as it can under restricted resources) and testing is then simply doing the rest.

Our technique is based on conditional model checking [7], which allows to save the information computed in a verification run (complete or incomplete) in the form of a so-called *condition*. In [7], this condition is given to a second, different verifier to complete verification. Here, we will use the condition to compute a residual program for a subsequently running testing tool. Two guiding principles lead the construction of residual programs: on the one hand, we do not want to test program parts which already have been verified w.r.t. the properties under interest, and on the other hand, we need to generate syntactically correct programs again such that these can be fed into standard testing tools. Here, we will propose two techniques for this, both fulfilling these guidelines, which have however different consequences for the testing step. Technique 1 uses the condition itself to generate a residual program, building the synchronous product of condition and original program thereby removing the already verified parts. This usually leads to a residual program which is structurally different from the original program. The second technique uses the condition to extract a *slice* of the original program (thus obtaining a syntactic subprogram) which is then used as residual program.

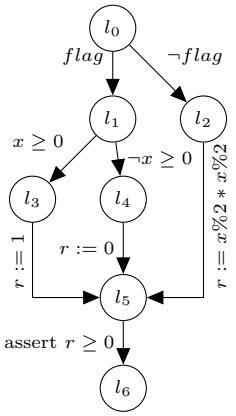
Our technique can be proven to be sound modulo soundness of the testing tool, i.e., if the testing tool could faithfully show absence of errors our combined technique would be able to definitely state safety of programs. We have implemented our technique using the software analysis tool CPAchecker [9] as verification tool and the concolic testing tool KLEE [10] for dynamic analysis. Using the results of our experiments we will also discuss which of the two residual program construction techniques is more suitable for which type of program.

2 Background

For the description of our approach, we assume programs to be written in a simple imperative language using assignments, assume and assert statements on integer variables only.[1] Following [8] describing configurable program analysis (the verification framework we employ later), we model programs as *control-flow automata* (CFA) $P = (L, G, l_0)$, where L is the set of control locations,

[1] In our experiments we use programs written in C intermediate language (CIL) [28].

```
l0 : if(flag)
l1 :   if(x>=0)
l3 :     r:=1;
       else
l4 :     r:=0;
     else
l2 :   r:=x%2*x%2;
l5 : assert r>=0;
l6 :
```




```
l0 : s:=0;
l1 : if(x<=0)
l2 :   assert s>=0;
     else
l3 :   i:=1;
l4 :   while(i<=x)
l5 :     s:=s+i;
l6 :     i:=i+1;
l7 :   assert s>0;
l8 :
```

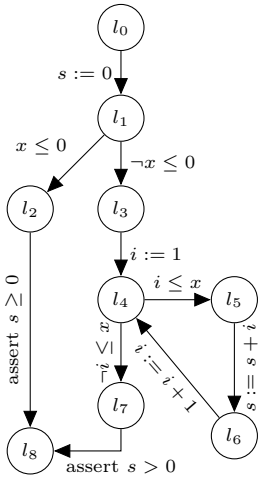


Fig. 1. Example programs `EVEN/SIGN` and `SUM` and their CFAs

$G \subseteq L \times Ops \times L$ the control flow edges and l_0 the program entry location. The set Ops contains all assignments, assume and assert statements. Figure 1 shows our example programs `EVEN/SIGN` and `SUM` and their CFAs. Program `EVEN/SIGN` computes – depending on the value of variable $flag$ – either the sign of variable x or whether x is even or odd. Program `SUM` sums up all integer values in interval $[0; x]$. The `assert` statement states the property to be checked. Indicator r describing if x is even/odd is expected to be non-negative. Sum s is non-negative and positive if $x > 0$. Note, the property (sum positive if $x > 0$) stated in line 7 of `SUM` is only true when neglecting overflows. Both CFAs contain two assume edges per condition of an if- or while-statement (one per valuation), and one assignment and assertion edge for each assignment and assertion, respectively.

The semantics of a program $P = (L, G, l_0)$ is given by a labeled transition system $T(P) = (C, G, \rightarrow)$ consisting of a set of concrete states C, the labels G

(the control-flow edges of the program) and a transition relation $\rightarrow \subseteq C \times G \times C$. We write $c \xrightarrow{g} c'$ for $(c, g, c') \in \rightarrow$. Let V denote the set of all integer variables of program P. A *concrete state* c either assigns to any variable $v \in V$ a value $c(v)$ and to the program counter a control location $c(pc) \in L$ or it is the error state, $c = c_{err}$, denoting that an assertion is violated. A transition $c \xrightarrow{(l,op,l')} c'$ is contained in $T(P)$ if $c \neq c_{err}$ and $c(pc) = l$ and either op is an assume statement, $c \models op$, $c'(pc) = l'$ and $\forall v \in V : c(v) = c'(v)$, or $op \equiv v := expr$ is an assignment and $c'(pc) = l'$, $c' = c[v \mapsto expr]$ or $op \equiv \texttt{assert}\ cond$ and either $c \models cond$ and $c'(pc) = l'$ and $\forall v \in V : c(v) = c'(v)$ or $c \not\models cond$ and $c' = c_{err}$. We call $c_0 \xrightarrow{g_0} c_1 \xrightarrow{g_1} \cdots \xrightarrow{g_{n-1}} c_n$ a *path* of program P if $c_0 \neq c_{err}$, $c_0(pc) = l_0$ and $\forall 0 \leq i < n : c_i \xrightarrow{g_i} c_{i+1}$. Intuitively, a path of a program describes a (partial) execution of P. We denote the set of all paths of P by $paths(P)$.

Finally, we are interested in program safety. In our case this means that none of program P's executions violates an assertion. All (partial) executions of P, that are all paths in $paths(P)$, are safe. Formally, a path $c_0 \xrightarrow{g_0} c_1 \xrightarrow{g_1} \cdots \xrightarrow{g_{n-1}} c_n \in paths(P)$ is *safe* if $c_n \neq c_{err}$. Let $paths_{safe}(P) \subseteq paths(P)$ denote the set of safe program paths. Then, a program P is *safe* if $paths_{safe}(P) = paths(P)$.

Unfortunately, a verification tool may fail to prove a program safe, e.g. due to resource limits it only proves that a subset of the program paths are safe. We use conditional model checking (CMC) [7] to describe which paths are proven safe by the verification tool and which paths still need to be verified. CMC can be used with any verification tool that keeps track of its (abstract) state space exploration in form of a reachability graph. Subtrees of the reachability graph which are completely verified are aggregated into a single safe state. For all other parts there is a one to one correspondence between the reachability graph and the *condition* generated by CMC. Coming back to our example programs `EVEN/SIGN` and `SUM` (Fig. 1), we assume that the verification tool only verified the left branch of the (outer) if statements. Figure 2 shows the conditions for these partial verifications. The rectangle node is the safe state. Since the left branch of each program has already been verified, it directly ends in the safe state. For the unproven right part of program `EVEN/SIGN` there is one automaton state per CFA location and if two of these CFA locations are connected by an edge g, then there is an edge between the corresponding automaton states and the label is the CFA edge g. For the unproven right part of program `SUM` we see that the verifier already revealed that the while loop is executed at least once and that it unrolled the while loop once (see path q_3, q_4, q_5, q_6).

Formally, a conditition can be defined as follows. For the details of the condition construction and the CMC approach we refer the reader to [7].[2]

Definition 1. *A* condition *for a program $P = (L, G, l_0)$ is a four-tuple $C_P = (Q, \delta, q_0, q_s)$, where Q is a set of states, $\delta \subseteq Q \times G \times Q$ a transition function, q_0 the initial state and q_s the safe state. The transition function ensures that the safe state is never left, i.e., $\forall g \in G : (q_s, g, q_s) \in \delta$. A* run *of C_P is a sequence of states $q_0 q_1 \ldots q_n$ such that $\forall 0 \leq i < n\ \exists (q_i, \cdot, q_{i+1}) \in \delta$.*

[2] Note that, the condition is called assumption automaton in [7].

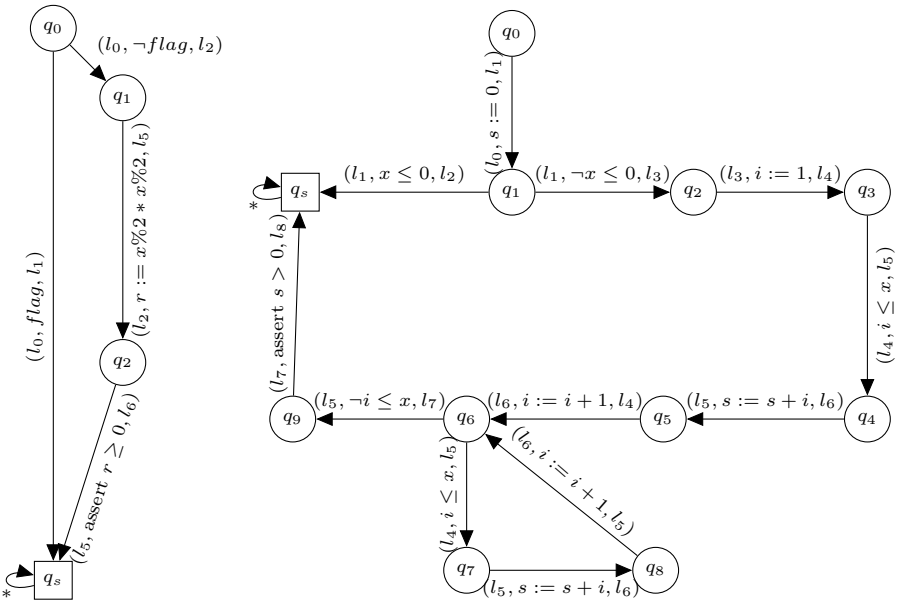


Fig. 2. Conditions showing partial verification result for programs `EVEN/SIGN` and `SUM`

We further assume that the verification tools produce conditions C_P for programs P fulfilling the following well-formedness properties. These properties ensure that the condition correctly summarizes the work done by the verification tool. Note, our verification tool CPAchecker always generates well-formed conditions.

Path Coverage. Every program path is described by a run of the condition. Formally, for all paths $c_0 \overset{g_0}{\to} c_1 \overset{g_1}{\to} \cdots \overset{g_{n-1}}{\to} c_n \in paths(P)$ there exists a run $q_0 q_1 \dots q_n$ s.t. $\forall 0 \leq i < n : (q_i, g_i, q_{i+1}) \in \delta$.

Safety. If the tail of a program path is subsumed by the safe state of the condition, then the program path is safe. Formally, all paths $c_0 \overset{g_0}{\to} c_1 \overset{g_1}{\to} \cdots \overset{g_{n-1}}{\to} c_n \in paths(P)$ are safe for which a run $q_0 q_1 \dots q_n$ exists s.t. $\forall 0 \leq i < n : (q_i, g_i, q_{i+1}) \in \delta$ and $\exists k < n \forall k \leq j \leq n : q_j = q_s$.

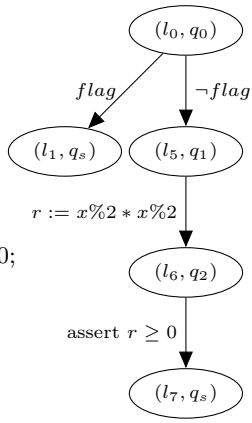
If a program has been completely verified and is safe, then the condition consists of two states only, namely q_0 and q_s. Given a condition describing which program paths are verified and which not, our idea is to provide to a test tool only the non-proven program paths in form of a residual program. Next, we describe two techniques to compute such a residual program, one based on subprogram extraction and the other on slicing.

3 Extraction of Residual Program from Condition

Our first technique extracts a subprogram from program P which contains only the unproven program paths. The idea is that the subprogram P' results from

(a) Residual program of `EVEN/SIGN`

```
(l0 ,q0): if(flag)
(l1 ,qs):   ;
          else
(l5 ,q1):   r:=x%2*x%2;
(l6 ,q2):   assert r>=0;
(l7 ,qs):
```



(b) Residual program of `SUM`

```
(l0 ,q0): s:=0;
(l1 ,q1): if(x<=0)
(l2 ,qs):   ;
          else
(l3 ,q2):   i:=1;
(l4 ,q3):   if(i<=x)
(l5 ,q4):     s:=s+i;
(l6 ,q5):     i:=i+1;
(l4 ,q6):   while(i<=x)
(l5 ,q7):     s:=s+i;
(l6 ,q8):     i:=i+1;
(l7 ,q9):   assert s>0;
(l8 ,qs):
```

Fig. 3. Source code of residual program constructed by synchronization of example programs and respective condition, also CFA for residual program of `EVEN/SIGN`

the syntactic, synchronous composition of program P and condition C_P. Subprogram P' starts in the initial locations of P and C_P and may only execute an operation if both P and C_P agree that the execution step is possible in the current situation. Furthermore, executions of P' stop if C_P determines that all possible extensions of the execution are already proven safe (C_P reaches safe state q_s). The subprogram extracted this way is the residual program used for further validation. Note, the residual program is not necessarily a syntactic subprogram. Its branching structure may be different. Nevertheless, all verified program executions are excluded from the residual program.

Figure 3 shows the source code for the residual programs extracted in the described way for our example programs (Fig. 1) and the respective conditions from Fig. 2. For program `EVEN/SIGN` also its CFA is given. The locations of the residual programs are a product of original program location and condition state. Moreover, the residual program contains an edge from location (l, q) to location (l', q') if an edge $g = (l, op, l')$ in the original program and a transition (q, g, q') in the condition exist. Furthermore, it stops in locations $(\cdot, q_s)$. Both residual programs remove the proven program part but since in the condition for program `SUM` the while loop is unrolled already once, the residual program of program `SUM` has more locations than program `SUM`. The following definition now formally describes the explained construction of the residual program.

Definition 2. *Let $P = (L, G, l_0)$ be a program and $C_P = (Q, \delta, q_0, q_s)$ a well-formed condition for P. The* residual program *extracted from P and C_P, denoted by $residual\_program\_of(P, C_P)$, is a program $P' = (L', G', l'_0)$ inductively defined as follows:*

1. $(l_0, q_0) \in L'$,
2. *if $(l, q) \in L'$, $q \neq q_s$, $g = (l, op, l') \in G$ and $(q, g, q') \in \delta$, then $(l', q') \in L'$ and $((l, q), op, (l', q')) \in G'$.*

Our goal is to validate the part of a program that has not yet been proven safe by the verification tool. Since we plan to check the residual program, we need a correspondence between safety of program P and residual program P'. Especially, the residual program P' may not lack any unsafe program path of P. To prevent the user from being bothered by non-existing bugs, the residual program P' should not contain new unsafe program paths that are not contained in program P. The following theorem guarantees these properties.

Theorem 1. *Let P be a program and C_P a well-formed condition for P. Then, program P is safe iff $residual\_program\_of(P, C_P)$ is safe.*

Proof (by contraposition). Denote $P' = residual\_program\_of(P, C_P)$
"$\Rightarrow$" If P is unsafe, exists $c_0 \xrightarrow{(l_0,op_0,l_1)} c_1 \xrightarrow{(l_1,op_1,l_2)} \dots \xrightarrow{(l_{n-1},op_{n-1},l_n)} c_{err} \in paths(P)$ and run $q_0 q_1 \dots q_n$ s.t. $\forall 0 \leq i < n : q_i \neq q_s \wedge (q_i, (l_i, op_i, l_{i+1}), q_{i+1}) \in \delta$ (C_P well-formed). Consider $p = c'_0 \xrightarrow{((l_0,q_0),op_0,(l_1,q_1))} c'_1 \xrightarrow{((l_1,q_1),op_1,(l_2,q_2))} \dots \xrightarrow{((l_{n-1},q_{n-1}),op_{n-1},(l_n,q_n))} c_{err}$ s.t. $\forall 0 \leq i < n : c'_i(pc) = (l_i, q_i) \wedge \forall v \in V : c_i(v) = c'_i(v)$. By construction of P' and definition $p \in paths(P')$. P' is unsafe.
"$\Leftarrow$" If P' is unsafe, exists a path $c_0 \xrightarrow{((l_0,q_0),op_0,(l_1,q_1))} c_1 \xrightarrow{((l_1,q_1),op_1,(l_2,q_2))} \dots \xrightarrow{((l_{n-1},q_{n-1}),op_{n-1},(l_n,q_n))} c_{err} \in paths(P')$. Now, consider $p = c'_0 \xrightarrow{(l_0,op_0,l_1)} c'_1 \xrightarrow{(l_1,op_1,l_2)} \dots \xrightarrow{(l_{n-1},op_{n-1},l_n)} c_{err}$ s.t. $\forall 0 \leq i < n : c'_i(pc) = l_i \wedge \forall v \in V : c_i(v) = c'_i(v)$. By construction of P' and definition $p \in paths(P)$. P is unsafe.

4 Residual Program via Slicing

Our second technique uses program slicing [32] to compute the residual program that describes the unproven part of the program. Program slicing is a technique for extracting those parts of a program which may affect a so-called *slicing criterion*. Slicing usually computes executable subprograms which is important since we want to give the residual program to a testing tool. We use slicing in the following way: First, we use the condition to identify those assertions of program P that have not been fully proven by the verification tool. Then, we take these assertions as slicing criteria to get those program parts of P which influence the unproven assertions. The obtained program slice is the residual program.

Next, we are coming to the details. First, we need to identify the set of unproven assertions. Given a well-formed condition C_P, we only know that assertions which at most occur in transitions of the form $(q_s, \cdot, q_s) \in \delta$ are not violated. Hence, any assertion `assert` *cond* that occurs in a transition $(q, (\cdot, \texttt{assert}\ cond, \cdot), q') \in \delta, q \neq q_s$ must be in the set of unproven assertions.[3] Looking at our example conditions (Fig. 2), we see that one unproven assertion in each condition, `assert` $r \geq 0$ and `assert` $s > 0$, respectively, exists.

[3] Our implementation represents `assert` *cond* by `if(¬cond) __assert_fail(...)`. We only add assertions if a transition with `__assert_fail(...)` exists. Thus, we do not add proven assertions which are on a program path that is not completely proven.

To distinguish between same assertions (same operation) used on different CFA edges, we use the CFA edge to describe an assertion. Hence, for our examples the sets of unproven assertions are $S_{C_{\texttt{EVEN/SIGN}}} = \{(l_6, \texttt{assert } r \geq 0, l_7)\}$ and $S_{C_{\texttt{SUM}}} = \{(l_7, \texttt{assert } s > 0, l_8)\}$. Generally, the set of unproven assertions is defined as follows.

Definition 3. *Let P be a program and $C_P = (Q, \delta, q_0, q_s)$ a well-formed condition for P. The* set S_{C_P} of unproven assertions *is defined as*

$$S_{C_P} = \{g \mid \exists (q, g, q') \in \delta \wedge q \neq q_s \wedge g \equiv (\cdot, \texttt{assert } \cdot, \cdot)\} \ .$$

To ensure that we do not miss any bug, we must assure that the computed set of unproven assertions S_{C_P} is complete. This means that if an unsafe path in the original program exists that violates assertion a, then a is contained in the set of unproven assertions. The following lemma gives us this property.

Lemma 1. *Let P be a program and C_P a well-formed condition for P. If P is unsafe, then an assertion $g \in S_{C_P}$ from the set of unproven assertion is violated.*

Proof (by contradiction). Let P be unsafe. By definition an unsafe program path exists. Let $p = c_0 \overset{g_0}{\rightarrow} c_1 \overset{g_1}{\rightarrow} \cdots \overset{g_{n-1}}{\rightarrow} c_{err} \in paths(P)$ be an arbitrary unsafe path. Since p unsafe, $g_{n-1} \equiv (\cdot, \texttt{assert } \cdot, \cdot)$. Assume $g_{n-1} \notin S_{C_P}$. Since C_P well-formed, exists run $q_0 q_1 \ldots q_n$ s.t. $\forall 0 \leq i < n : (q_i, g_i, q_{i+1}) \in \delta$. Since $g_{n-1} \notin S_{C_P}$, it follows that $q_{n-1} = q_n = q_s$. Contradiction to well-formedness (safety) of C_P.

After computation of the set S_{C_P} of unproven assertions from condition C_P, we now generate the residual program via slicing. The general idea of slicing is to delete those statements from the program that do not influence the semantic property defined by the slicing criteria. Typical slicing criteria are the variable values at certain program location. We are interested in the evaluation of the computed, unproven assertions at the location they are defined. Slicing should delete those statements which do not influence evaluation of any assertion in S_{C_P}. Looking at our example programs (Fig. 1) and the sets of unproven assertions $S_{C_{\texttt{EVEN/SIGN}}} = \{(l_6, \texttt{assert } r \geq 0, l_7)\}$ and $S_{C_{\texttt{SUM}}} = \{(l_7, \texttt{assert } s > 0, l_8)\}$, we see that we cannot delete any statement in program `EVEN/SIGN`. Every statement influences the evaluation of the assertion. In program `SUM` only the right branch of the if statement influences the assertion in $S_{C_{\texttt{SUM}}}$. Hence, slicing deletes the statements of the left branch. The resulting program slice, the residual program, for `SUM` and $S_{C_{\texttt{SUM}}} = \{(l_7, \texttt{assert } s > 0, l_8)\}$, is the residual program shown in Fig. 4. Here, we refrain from defining the computation of program slices but just define constraints on the constructed slice, which standard slicing technique will however give us. Slicing only removes program statements (CFA edges). Technically, a statement is removed by deleting the respective CFA edge (l, op, l'). To keep the initial location l_0, we do not relink l's predecessors. Instead, we relink successors of l' to l. Furthermore, locations without predecessors and successors are removed. The following definition describes the structural appearance of a program slice obtained from a program by deletion of some the program's statements.

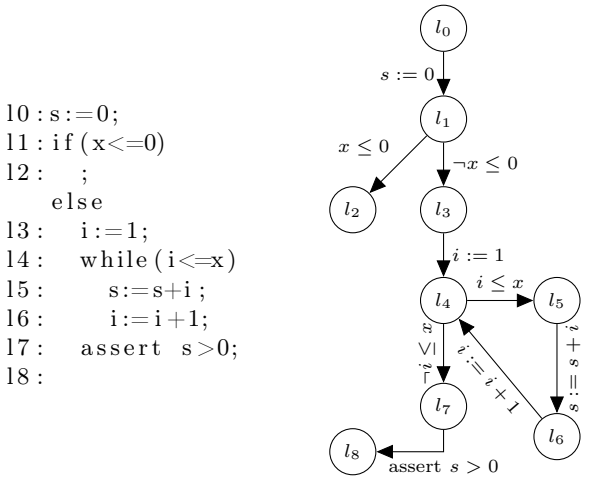


Fig. 4. C code and CFA of residual program constructed by slicing of program `SUM` using slicing criterion $S_{C_{\texttt{SUM}}} = \{(l_7, \texttt{assert } s > 0, l_8)\}$

Definition 4. *Let $P = (L, G, l_0)$ be a program. A* slice of P *is a program $P' = (L', G', l_0')$ with $L' \subseteq L$, $l_0' = l_0$ and if $(l, op, l') \in G'$, then a sequence of locations $l_1 \dots l_n$, $n \geq 2$, exists with $l_1 = l$, $l_n = l'$, $(l_{n-1}, op, l') \in G$, $\forall 1 \leq i < n-1 : (l_i, \cdot, l_{i+1}) \in G \wedge l_i \notin L'$.*

So far, we defined the structural appearance of a program slice. In addition we require that the behavior of program slice and original program is identical w.r.t. the slicing criterion. In general, a slicing criterion is a set of program statements of interest. Since program statements are defined by CFA edges and we are only interested in unproven assertions, the slicing criterion is a set of assertion edges, namely the set of unproven assertions. In this case, original program and a program slice behave identically w.r.t. the slicing criterion – we call the program slice sound w.r.t. the slicing criterion – if the following holds. If an arbitrary execution of the original program violates an assertion g_a in the slicing criterion, then an execution of the program slice exists, that violates the corresponding assertion g_a' in the program slice. Formally, this is stated as follows.

Definition 5. *Let $P = (L, G, l_0)$ be a program and SC a slicing criterion, $SC \subseteq G_{\texttt{assert}} = \{g \mid g \in G \wedge g \equiv (\cdot, \texttt{assert } \cdot, \cdot)\}$. Then, a program slice $P' = (L', G', l_0)$ of program P is* sound *w.r.t. the slicing criterion SC if for any concrete state $c_0 \in C$ the following holds: if there is a path $c_0 \xrightarrow{g_0} c_1 \cdots \xrightarrow{g_{n-1}} c_{err} \in paths(P) \wedge g_{n-1} = (l, \texttt{assert } expr, l') \in SC$ in program P, then there is also a path $c_0 \xrightarrow{g_0'} c_1' \cdots \xrightarrow{g_{m-1}'} c_{err} \in paths(P') \wedge g_{m-1}' = (\cdot, \texttt{assert } expr, l')$ in program slice P'.*

We use dependence based slicing [23] to compute a program slice w.r.t a slicing criterion SC. The work in [2] ensures that the computed slice obtained from

dependence based slicing is sound w.r.t. SC. That is why, in the latter we assume that the residual program, the program slice computed from program P and slicing criterion S_{C_P} is sound w.r.t. S_{C_P}.

Remember, our goal is the validation of the non-proven part of the original program. Validating the residual program obtained from slicing the original program using the set of unproven assertions as slicing criterion is planned as one option. That is why, we need some correspondence between the original program and this residual program. Especially, the residual program must only be safe if the original program is safe. The following theorem ensures this property

Theorem 2. *Let P be a program, C_P a well-formed condition for P and P' a slice of program P which is sound w.r.t. the set of unproven assertions S_{C_P} (the slicing criterion). P is safe if P' is safe.*

Proof (by contraposition). If P is unsafe, exists a path $c_0 \xrightarrow{(l_0,op_0,l_1)} c_1 \xrightarrow{(l_1,op_1,l_2)} \ldots \xrightarrow{(l_{n-1},op_{n-1},l_n)} c_{err} \in paths(P)$ and $op_{n-1} \equiv \texttt{assert}\ cond$. Due to lemma 1 $(l_{n-1}, op_{n-1}, l_n) \in S_{C_P}$. Since slice P' is sound, P' is unsafe.

Note that the two presented residual program construction techniques can be combined as follows. Given a program P and a condition C_P for P, first, use the technique extraction of residual program from condition and compute program $P' = (L', G', l'_0) = residual\_program\_of(P, C_P)$. Take all assertions in P' as unproven assertions, $S_{C'_P} = \{g \mid g \in G' \wedge g \equiv\ (\cdot, \texttt{assert}\ \cdot, \cdot)\}$, and apply the technique residual program via slicing to get program slice P'' sound w.r.t. $S_{C'_P}$. Program P'' is the input for testing in the combined approach.

5 Experimental Results

In our experiments, we studied the actual benefits of our techniques for combined verification and testing. For that, we examined whether a partial verification run with subsequent testing is faster than a complete verification. Furthermore, we determined whether a residual program reduces the test effort. Finally, we compared all three techniques, the two techniques for construction of residual programs as well as their combination, with the naive approach always testing the complete program.

For (partial) verification we used the configurable software analysis tool CPAchecker (svn r13520). We configured CPAchecker to use predicate analysis[4] and to produce a condition after partial verification. Also, the residual program from condition is generated by CPAchecker. We sliced the original program and the residual program from condition with the help of the source code analysis platform Frama-C (v.Neon-20140301) [17]. Finally, we utilized the concolic test tool KLEE (v.git-20140327) [10] to generate the test-case suites. We evaluated our techniques on our examples, on two programs called `search`

[4] Our technique allows the usage of arbitrary analyses, e.g. value analysis or even sequential combinations of analyses as illustrated in [7].

Table 1. Experimental Results

	Verification		Verification+Testing				Program Size				#Tests			
Program	V	p.V	P	C	S	C+S	P	C	S	C+S	P	C	S	C+S
`EVEN/SIGN`	2.71	2.71	**2.8**	3.15	3.45	3.59	51	43	51	**34**	3	**2**	3	**2**
`SUM`	3.66	3.48	**4.54**	4.68	5.52	5.64	49	51	**48**	49	2	3	**1**	3
search	7.55	4.17	5.51	5.73	5.73	**5.29**	217	243	201	**167**	14	17	14	**8**
sort	16.28	11.23	**11.88**	12.94	12.32	13.61	760	780	**569**	574	19	19	15	**14**
get_tag	9.71	4.11	4.71	**4.64**	5.1	5.35	415	248	**159**	199	16	8	**6**	8
mim7to8	21.58	14.34	20.8	23.18	**17.07**	20.16	939	1142	**538**	966	48	79	**28**	48
esc_uri	72.07	49.91	**50.56**	52.5	51.35	53.61	808	1052	**638**	757	29	51	**26**	38

and `sort`, which allow the user to select one from popular array search and array sort algorithms, as well as 3 programs constructed from the Verisec benchmark [26]. Each of the three programs looks at one test case in the benchmark and allows to select between the different variants of that test case available in the benchmark. Except for our examples, all assert statements checked the absence of buffer-overflows. Furthermore, every program was preprocessed with CIL [28]. All programs are error free excluding overflows which the verification tool assumes not to happen. Moreover, the test tool found the errors regarding overflows if they are part of the test tool's input program. Our experiments were performed on a 2.4 Ghz Intel Core 2 Duo Arch Linux system with 4GB memory and statement coverage, measured with gcov from the GNU Compiler Collection (v.4.8.2) [1], was always about 90-100% for all generated test-case suites.

Table 5 shows our results. All times are given in seconds. Columns V and p.V show the times for complete and partial verification. The times for `EVEN/SIGN` are the same because due to the modulo operator used in `EVEN/SIGN` CPAchecker was not able to prove the full program using predicate analysis. The next four columns show the total times for partial verification and subsequent testing for all four approaches: testing complete program (P), residual program from condition (C), residual program constructed via slicing (S) and the combination of the latter two (C+S). Total time includes the time for partial verification, construction of residual program (if any, e.g. slicing time) and test generation time. The last 8 columns reflect the test effort. First, the sizes of the programs put into the test tool are given in number of control locations. Thereafter, the number of generated test cases per test suite is depicted. The smallest total time, program size and number of tests for each evaluated benchmark program is presented in bold.

Our experiments show that in most cases, partial verification with subsequent testing is faster than a complete verification and thus, enables "on-the-fly" analysis. Surprisingly, the naive approach testing always the complete program performs best in half of the cases. We believe that this is not a general weakness of our approaches. In fact, KLEE performs well even for large programs (much larger than ours) and currently constructing the residual program takes a significant amount of time. The following comparison of program size and number of tests

will support our position. In contrast to total execution times, program size and number of tests are usually lower for the two residual program approaches based on slicing (S,C+S) than for the complete program but this is rarely true for the residual program from condition (C). In case of the slicing approaches, testing following a partial verification benefits from residual program construction. Moreover, none of the two slicing approaches (S, C+S) always outperforms the other. As already seen for our example programs `EVEN/SIGN` and `SUM` which approach performs better depends on the partial verification, e.g. do assertions exist which are only partially verified or do loops exist which are only partially verified. Hence, strategies choosing the correct technique need to be developed considering e.g. the program structure and the condition.

6 Related Work

There are a number of different approaches combining verification and testing which we shortly discuss here. VART [29] uses testing to identify likely invariants and exclude intentionally invalidated invariants after update. Bounded model checking determines the invariants from the likely invariants and checks the non-invalidated invariants on the updated program. In unit checking [20] a LTL formula specifies program paths being suspicious. A model checker explores these paths. A satisfying assignment to the respective path condition is used for test generation. Approaches like [31,30] check if a model satisfies a property and then verify that the implementation is consistent with the model.

Like us, many approaches use collaboration of verification and testing to either verify a program or find bugs. The approaches in [25,33,19,3,13,21,27] describe (interleaved) collaboration of verification and testing in which testing assists to find a proof. [33] generalizes test observations to a likely abstraction. A theorem prover checks the abstraction. If this check fails, the counterexample is used to compute the next input for testing and generalization starts again. [27] uses testing to choose a good abstraction configuration for the analysis. [21] uses test data to simplify invariant constraints. [25,19,3,13] including Synergy [19] and Dash [3] search for errors and proofs at the same time. Test information (e.g. unavailability of concrete counterexample) is used for refinement of abstraction in case of spurious abstract counterexamples. Abstract counterexamples are used to derive a test for a concrete counterexample. In our approaches collaboration is purely sequential, first verification, then testing, and testing does not assist verification.

A different class of collaborating approaches [6,15,16,18,12] uses static analysis to detect potential bugs and then test if the bugs really exist in the program. DyTa [18] first detects potential defects with a static analyzer, then identifies branching conditions which must be valid to trigger a defect, and adds an assume statement for this condition to guide test input generation via symbolic execution. BLAST [6] reports error locations considered reachable and computes a test input for every error path, a satisfying assignment for the symbolic error path. Check'n'Crash [15] and DSD-Crasher [16] use ESC/Java to identify

potential errors and compute a satisfying assignment to the error condition for every potential error. These assignments are given to JCrasher to generate JUnit tests. SANTE [12] uses a static abstract interpretation based value analysis to compute alarms. Then, it instruments the program with special error branches to enable alarm guided test generation and computes one or more program slices, depending on the configuration. These program slices are subject to test generation. Our second approach can be understood as testing potential defects. Any unproven assertion is a potential defect. In contrast to all the other approaches, we do not verify the whole program and we believe that an error found by our verification tool is a real bug. Nevertheless, similar to SANTE [12] in configuration ALL (computing a single slice) we also apply slicing using the non-verified assertions instead of the alarms as slicing criteria.

Further approaches [24,7,14] divide the program like we do. One part of the program is verified, the other is tested. Program partitioning [24] takes the opposite direction. It first tests. Then, it removes the sufficiently tested paths and verifies the residual program. In contrast to our approaches the residual program in [24] is always a subgraph of the original one. Conditional model checking [7] produces an assumption automaton to sum up the verification work and thus partitions the program. In [7] the technique is used to verify the non-verified partition by a second verifier. It is only mentioned that the technique can be used to guide test generation. We describe two approaches for actual guidance. Christakis et al. [14] instrument the program with assumptions and assertions describing the verification effort already done. The subsequent testing tool is guided to test the assumptions or the validated property but in contrast to our approaches, the tested program is not simplified by slicing or deletion of program paths.

Conditioned program slicing [11] and its extension [22] provide a general model for the extraction of those program parts which keep the behavior of a program statement w.r.t. a set of program executions. Hence, their idea is similar to our extraction of a residual program from condition and our combined approach. An important difference between our approaches and [11,22] is that we use a structural description of the program executions (the condition) and [11,22] require a logic formula on a subset of the input variables. We think that it is non-trivial to transform our condition into a logic formula needed for conditioned program slicing.

7 Conclusion

In this paper we presented a new way of combining verification and testing. The approach divides the labor of safety checking onto verification and testing, testing only having to analyze those parts of the program which have not been verified. To this end, we presented two ways of constructing residual programs for testing. We implemented both techniques and experimentally evaluated them on a number of example programs. The experiments showed that in almost all

cases the subsequent testing following an incomplete verification could benefit from residual program construction.

As future work we in particular would like to work on strategies for dividing the available amount of time into the part for verification and that for testing. Furthermore, the experiments so far indicate that the program's syntactical structure might influence what residual program construction technique works better, so that the choice for the technique could be based on the program at hand. Another line of improvement could be a parallelization of the two techniques since they are completely independent.

References

1. Gnu compiler collection, `https://gcc.gnu.org` (accessed: October 13, 2014)
2. Barraclough, R.W., Binkley, D., Danicic, S., Harman, M., Hierons, R.M., Kiss, Á., Laurence, M., Ouarbya, L.: A trajectory-based strict semantics for program slicing. Theoretical Computer Science 411(11-13), 1372–1386 (2010)
3. Beckman, N.E., Nori, A.V., Rajamani, S.K., Simmons, R.J.: Proofs from tests. In: ISSTA 2008, pp. 3–14. ACM (2008)
4. Bertolino, A.: Software testing research: Achievements, challenges, dreams. In: Briand, L.C., Wolf, A.L. (eds.) International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, Minneapolis, MN, USA, May 23-25, pp. 85–103 (2007)
5. Beyer, D.: Status report on software verification. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014 (ETAPS). LNCS, vol. 8413, pp. 373–388. Springer, Heidelberg (2014)
6. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating tests from counterexamples. In: ICSE 2004, pp. 326–335. IEEE Computer Society (2004)
7. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: A technique to pass information between verifiers. In: FSE 2012, pp. 1–11. ACM (2012)
8. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: Concretizing the convergence of model checking and program analysis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 504–518. Springer, Heidelberg (2007)
9. Beyer, D., Keremoglu, M.E.: CPAchecker: A Tool for Configurable Software Verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011)
10. Cadar, C., Dunbar, D., Engler, D.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI 2008, pp. 209–224. USENIX Association (2008)
11. Canfora, G., Cimitile, A., De Lucia, A.: Conditioned program slicing. Information and Software Technology 40(11-12), 595–607 (1998)
12. Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand, J.: Program slicing enhances a verification technique combining static and dynamic analysis. In: SAC 2012, pp. 1284–1291. ACM (2012)
13. Chen, J., MacDonald, S.: Towards a better collaboration of static and dynamic analyses for testing concurrent programs. In: PADTAD 2008, pp. 8:1–8:9. ACM (2008)

14. Christakis, M., Müller, P., Wüstholz, V.: Collaborative verification and testing with explicit assumptions. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 132–146. Springer, Heidelberg (2012)
15. Csallner, C., Smaragdakis, Y.: Check 'N' Crash: Combining static checking and testing. In: ICSE 2005, pp. 422–431. ACM (2005)
16. Csallner, C., Smaragdakis, Y.: DSD-Crasher: A hybrid analysis tool for bug finding. In: ISSTA 2006, pp. 245–254. ACM (2006)
17. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 233–247. Springer, Heidelberg (2012)
18. Ge, X., Taneja, K., Xie, T., Tillmann, N.: DyTa: Dynamic symbolic execution guided with static verification results. In: ICSE 2011, pp. 992–994. ACM (2011)
19. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: SYNERGY: a new algorithm for property checking. In: SIGSOFT FSE 2006, pp. 117–127. ACM Press (2006)
20. Gunter, E., Peled, D.: Model checking, testing and verification working together. Formal Aspects of Computing 17(2), 201–221 (2005)
21. Gupta, A., Majumdar, R., Rybalchenko, A.: From tests to proofs. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 262–276. Springer, Heidelberg (2009)
22. Harman, M., Hierons, R., Fox, C., Danicic, S., Howroyd, J.: Pre/post conditioned slicing. In: ICSM 2001, pp. 138–147 (2001)
23. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. In: PLDI 1988, pp. 35–46. ACM (1988)
24. Jalote, P., Vangala, V., Singh, T., Jain, P.: Program partitioning: A framework for combining static and dynamic analysis. In: WODA 2006, pp. 11–16. ACM (2006)
25. Kroening, D., Groce, A., Clarke, E.: Counterexample guided abstraction refinement via program execution. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 224–238. Springer, Heidelberg (2004)
26. Ku, K., Hart, T.E., Chechik, M., Lie, D.: A buffer overflow benchmark for software model checkers. In: ASE 2007, pp. 389–392. ACM (2007)
27. Naik, M., Yang, H., Castelnuovo, G., Sagiv, M.: Abstractions from tests. In: POPL 2012, pp. 373–386. ACM (2012)
28. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: Nigel Horspool, R. (ed.) CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002)
29. Pastore, F., Mariani, L., Hyvärinen, A.E.J., Fedyukovich, G., Sharygina, N., Sehestedt, S., Muhammad, A.: Verification-aided regression testing. In: ISSTA 2014, pp. 37–48. ACM (2014)
30. Rusu, V., Marchand, H., Tschaen, V., Jéron, T., Jeannet, B.: From safety verification to safety testing. In: Groz, R., Hierons, R.M. (eds.) TestCom 2004. LNCS, vol. 2978, pp. 160–176. Springer, Heidelberg (2004)
31. Sharygina, N., Peled, D.: A combined testing and verification approach for software reliability. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 611–628. Springer, Heidelberg (2001)
32. Tip, F.: A survey of program slicing techniques. Journal of Programming Languages 3(3) (1995)
33. Yorsh, G., Ball, T., Sagiv, M.: Testing, abstraction, theorem proving: Better together? In: ISSTA 2006, pp. 145–156. ACM (2006)