Software Partitioning for Effective Automated Unit Testing

Arindam Chakrabarti*
University of California at Berkeley
arindam@cs.berkeley.edu

Patrice Godefroid
Bell Laboratories, Lucent Technologies
god@bell-labs.com

ABSTRACT

A key problem for effective unit testing is the difficulty of partitioning large software systems into appropriate units that can be tested in isolation. We present an approach that identifies control and data inter-dependencies between software components using static program analysis, and divides the source code into units where highly-intertwined components are grouped together. Those units can then be tested in isolation using automated test generation techniques and tools, such as dynamic software model checkers. We discuss preliminary experimental results showing that automatic software partitioning can significantly increase test coverage without generating too many false alarms caused by unrealistic inputs being injected at interfaces between units.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Verification, Algorithms, Reliability

Keywords

Software Testing and Model Checking, Compositional Analysis, Interfaces, Program Verification

1. INTRODUCTION

Today, testing is the primary way to check the correctness of software. Correctness is even more important to determine in the case of embedded software, where reliability and security are often key features. Billions of dollars are spent every year on testing in the software industry as a whole as testing usually accounts for about 50% of the cost of software development [29]. Yet software failures are numerous and their cost to the world's economy can also be counted in billions of dollars [31].

An approach (among others) that has potential to significantly improve on the current state of the art consists of automatically generating test inputs from a static or dynamic program analysis in order to force program executions towards specific code statements. *Automated test generation* from program analysis is an old idea (e.g., [25, 29, 26, 13]), which arguably has had a limited practical impact so far, perhaps due to the lack of usable, industrial-strength tools implementing this approach. Recently, there has been a renewed interest for automated test generation (e.g., [6, 5, 36, 11, 17, 10]). This renewed interest can perhaps be attributed to recent progress on software model checking, efficient theorem proving, static analysis and symbolic execution technology, as well as recent successes in engineering more practically-usable static-analysis tools (e.g., [9, 20]) and the increasing computational power available on modern computers.

A new idea in this area is Directed Automated Random Testing (DART) [17], which fully automates software testing by combining three main techniques: (1) *automated* extraction of the interface of a program with its external environment using static source-code parsing; (2) automatic generation of a test driver for this interface that performs *random* testing to simulate the most general environment the program can operate in; and (3) dynamic analysis of how the program behaves under random testing and automatic generation of new test inputs that *direct* the execution along alternative program paths. DART can detect standard errors such as program crashes, assertion violations, and non-termination, and can also be used in conjunction with complementary run-time checking tools for detecting memory allocation problems (e.g., [22, 30, 27, 34]). During testing (Step 3), DART performs a *directed search*, a variant of dynamic test generation (e.g., [26, 19]). Starting with a random input, a DART-instrumented program calculates during each execution an input vector for the next execution. This vector contains values that are the solution of *symbolic constraints* gathered from predicates in branch statements during the previous execution. The new input vector attempts to force the execution of the program through a new path. By repeating this process, a directed search attempts to force the program to sweep through all its feasible ex-

*The work of this author was done mostly while visiting Bell Laboratories.

EMSOFT'06, October 22–25, 2006, Seoul, Korea.

ecution paths, in a style similar to *systematic testing* and *dynamic software model checking* [15].

When applied to large programs, a directed search is typically incomplete due to combinatorial explosion and because of the presence of complex program statements or external function calls for which no symbolic constraint can be generated. It is worth emphasizing that these two limitations – scalability and limited symbolic-reasoning capabilities – are inherent to *all* automated test-generation techniques and tools, whether static (e.g., [6, 5, 36, 11]) or dynamic (e.g., [17, 10]). This explains why *automated test generation typically cannot achieve 100% code coverage* in practice, independently of how code coverage is defined (e.g., coverage of all program statements, branches, paths, etc.).

An idea to alleviate these limitations is to partition a large program into several smaller units that are then tested in isolation to search for program bugs such as crashes and assertion violations. As an extreme, every function in a program could be tested one-by-one in isolation using automated test generation. While this testing strategy can dramatically increase code coverage, automated test generation will also typically generate lots of unrealistic inputs which in turn will trigger many unrealistic behaviors and spurious bugs, i.e., *false alarms*. This is because function testing ignores all the dependencies (preconditions implied by the possible calling contexts) between functions.

In this paper, we investigate *how to partition a program* into a set of units such that testing those units in isolation *maximizes overall code coverage while minimizing the number of false alarms*. We propose a two-step heuristic approach to this problem. First, we perform a light-weight static source-code analysis to identify the *interface* of every function in a program. Second, we divide the program into units where *highly-intertwined functions are grouped together*, hence hiding complex interfaces inside units. We have developed a prototype tool implementing these ideas for the C programming language. We present results of experiments performed on oSIP, an open-source implementation of the Session Initiation Protocol *embedded* in many IP phones. Those experiments show that automatic software partitioning using the above simple strategy can significantly increase test coverage without generating too many false alarms.

The rest of this paper is organized as follows. We start in Section 2 with some background definitions and assumptions, and define the software partitioning problem in this context. The two steps of our approach to the partitioning problem are then discussed in Sections 3 and 4, on interfaces and partitioning algorithms respectively. The partitioning algorithms of Section 4 have been implemented in a prototype tool for partitioning C programs, and results of experiments are presented in Section 5. Other related work is discussed in Section 6 and we conclude in Section 7.

2. THE SOFTWARE PARTITIONING PROBLEM

A program P is defined as a set of functions $\mathcal{F}$ associated with a given function *main* $\in \mathcal{F}$ which is always executed first when running the program. A *call graph* $\mathcal{G}$ over a set of *functions* $\mathcal{F}$ is a graph $\mathcal{G} = (V, E)$ with a set of *vertices* $V = \mathcal{F}$ and a set of *edges* $E \subseteq V \times V$ such that $(f, g) \in E$ if there is a call to function g in function f. By transitive closure, we say that a function f *calls* a function g in a call graph $\mathcal{G}$ if there is a sequence $\sigma = s_0, s_1, \ldots, s_k$ such that $s_0 = f$, and $s_k = g$, and $(s_i, s_{i+1}) \in E$ for $i \in \{0, 1, \ldots, k-1\}$; the *path* from f to g is the sequence σ, and its *length* is k. A function f *directly calls* a function g in $\mathcal{G}$ if there is a path from f to g of length 1 in $\mathcal{G}$.

A path $\sigma = s_0, s_1, \ldots, s_k$ from f to g *lies in* a set S if $s_i \in S$ for all $i \in \{0, 1, \ldots, k\}$. A set of functions S is *convex* with respect to a function $f \in S$ if for every function g in S called by f, there is a path from f to g that lies in S. Given a set S of functions, a function $f \in S$ is called an *entry point* of S if every function $g \in S$ is called by f, and S is convex with respect to f. A *unit* $u = (S, f)$ is a set of functions S with an entry point f.

Given a call graph $\mathcal{G}$ over a set of functions $\mathcal{F}$, a *partition* $\mathcal{P}$ of $\mathcal{G}$ is a set of units $\mathcal{P} = \{(S_0, f_0), (S_1, f_1), \ldots, (S_k, f_k)\}$ such that $\bigcup\{S_i \mid 0 \leq i \leq k\} = \mathcal{F}$, and $S_i \cap S_j = \emptyset$ for all $0 \leq i, j \leq k$ such that $i \neq j$. We assume that the pair ($\mathcal{F}$, *main*) is a unit, i.e., *main* is an entry point of $\mathcal{F}$. Trivially, for any function f, $(\{f\}, f)$ is always a unit.

In order to test program P, automated test generation techniques and tools can be applied to generate inputs at the program interface in order to drive its execution along as many program statements/branches/paths as possible. However, testing large programs is difficult because of the limited reasoning capabilities of automated test generation techniques and expensive because of the prohibitive number of statements, branches and paths in large programs. Consequently, viewing the entire program P as a single unit usually leads to poor code coverage (as will be shown experimentally later) if P is large and non-trivial. Let us call this form of testing *monolithic testing*.

On the other hand, automated test generation can also be applied to all the individual functions $f \in \mathcal{F}$ of P, one by one. This strategy can dramatically increase code coverage but automated test generation will typically generate lots of unrealistic inputs which in turn will trigger many unrealistic behaviors and hence many false alarms. Indeed, such a *piecemeal testing* ignores all the dependencies (preconditions implied by the possible calling contexts) between functions. For instance, a function f taking as input argument a pointer to a data structure may assume that this pointer is always non-null. While this assumption may be true whenever f is called by some function in $\mathcal{F}$, piecemeal testing may very well provide a null value for this argument when calling f, which may lead to a crash if the pointer is dereferenced inside f. We call such a spurious error a *false alarm*.

We define the *software partitioning problem* as follows:

> how to partition a given program P satisfying the above assumptions into a set of units such that testing those units in isolation maximizes code coverage while minimizing the number of false alarms?

Our definition is intentionally general to accommodate various specific ways to measure code coverage or identify/define false alarms.

In this paper, we investigate a two-step heuristic approach to the software partitioning problem. First, we propose to perform a light-weight static source-code analysis to identify the *interface* of every function in a program P, which enables the definition and measurement of the *complexity* of those interfaces. Second, we discuss several clustering algorithms which group together functions whose joint interface is complex, hence hiding that interface inside the resulting unit. The result is a partition of the program P where highly intertwined functions are grouped together in units.

The two steps of our approach are discussed in the next two sections.

3. INTERFACES

In principle, the interface of a function describes all possible avenues of exchange of information between the function and its environment: arguments, return values, shared variables, and calls to other functions. The interface of a unit is defined as the interface of the composition of the functions in that unit.

In practice, the precise interface of functions described in full-fledged programming languages like C or C++ can be hard to determine statically due to unknown control flow (e.g., in the presence of function pointers), unbounded data structures, side-effects through global variables, etc. We therefore use approximate interface representations.

Control interfaces track control dependencies across function and unit boundaries. Given a program P defined by a set $\mathcal{F}$ of functions, the *boolean control interface* $\mathcal{C}$ of a unit $u = (S, f_S)$ of P is a tuple $(S, \mathcal{O}, \mathcal{I})$ where $\mathcal{O} : S \times (\mathcal{F} \setminus S)$ is a relation mapping every function $f \in S$ to the functions $g \in \mathcal{F} \setminus S$ that are directly called by f, and $\mathcal{I} : (\mathcal{F} \setminus S) \times S$ is a relation mapping every function $g \in \mathcal{F} \setminus S$ to the functions $f \in S$ directly called by g.

By extension, given a program P defined by a set $\mathcal{F}$ of functions, the *weighted control interface* $\mathcal{C}_w$ of a unit $u = (S, f_S)$ of P is a tuple $(S, \mathcal{O}_w, \mathcal{I}_w)$ where $\mathcal{O}_w : S \times ((\mathcal{F} \setminus S) \times \mathbb{N})$ is a relation mapping every function $f \in S$ to the pairs (g, n) where $g \in \mathcal{F} \setminus S$ is a function directly called by f, and n is the number of calls to g in the code describing f, and $\mathcal{I}_w : (\mathcal{F} \setminus S) \times (S \times \mathbb{N})$ is a relation mapping every function $g \in \mathcal{F} \setminus S$ to the pairs (f, n) where $f \in S$ is a function directly called by g and n is the number of calls to f in the code describing g.

Two boolean (weighted) control interfaces $\mathcal{C}_1 = (S_1, \mathcal{O}_1, \mathcal{I}_1)$ and $\mathcal{C}_2 = (S_2, \mathcal{O}_2, \mathcal{I}_2)$ are *compatible* (denoted $comp(\mathcal{C}_1, \mathcal{C}_2)$) if $S_1 \cap S_2 = \emptyset$. Given two compatible boolean control interfaces $\mathcal{C}_1 = (S_1, \mathcal{O}_1, \mathcal{I}_1)$ and $\mathcal{C}_2 = (S_2, \mathcal{O}_2, \mathcal{I}_2)$, their *composition* (denoted $\mathcal{C}_1 \parallel \mathcal{C}_2$) is the boolean control interface $\mathcal{C}_c = (S_c, \mathcal{O}_c, \mathcal{I}_c)$ where $S_c = S_1 \cup S_2$, $\mathcal{O}_c = \{(f, g) \in \mathcal{O}_1 \cup \mathcal{O}_2 | g \in \mathcal{F} \setminus (S_1 \cup S_2)\}$ and $\mathcal{I}_c = \{(f, g) \in \mathcal{I}_1 \cup \mathcal{I}_2 | f \in \mathcal{F} \setminus (S_1 \cup S_2)\}$ (calls from S_1 and S_2 to each other are hidden by the composition). Similarly, the composition of two compatible weighted control interfaces is defined as the weighted control interface $\mathcal{C}_c = (S_c, \mathcal{O}_c, \mathcal{I}_c)$ where $S_c = S_1 \cup S_2$, $\mathcal{O}_c = \{(f, (g, n)) \in \mathcal{O}_1 \cup \mathcal{O}_2 | g \in \mathcal{F} \setminus (S_1 \cup S_2)\}$ and $\mathcal{I}_c = \{(f, (g, n)) \in \mathcal{I}_1 \cup \mathcal{I}_2 | f \in \mathcal{F} \setminus (S_1 \cup S_2)\}$.

Richer interface representations can be defined by also taking into account other features of a function that are externally visible, such as the number of input arguments, the type of input arguments and return values for incoming or outgoing function calls, the sequencing of external calls made by a function (flow-sensitive interface representation), etc.

For simplicity, we consider in this work partitioning algorithms that make use of information exposed by control interfaces only. Whether richer interface representations can lead to significantly better software partitioning is left for future work.

Algorithm 1 PartitionCP(S)

Input: A set of control interfaces of a set S of functions
Output: A partition of S into a set U of units
Variables: *WeightedGraph* G

$G := \mathit{PopularityGraph}(S)$
while $(\neg\ \mathit{IsEmpty}(G))$ **do**
  $c := \mathit{ChooseCPCutoff}(G)$
  $G' := \mathit{FilterPopularEdges}(G, c)$
  while $(\neg\ \mathit{IsEmpty}(G'))$ **do**
    $t := \mathit{TopLevel}(G')$
    $u := \mathit{Reachable}(G', t)$
    if $(|u| > 1$ **or** $G' = G)$ **then**
      add u as a new unit in U
      $G := \mathit{RemoveNodes}(G, u)$
    end if
    $G' := \mathit{RemoveNodes}(G', u)$
  end while
end while

4. SOFTWARE PARTITIONING ALGORITHMS

In this section, we present partitioning algorithms that exploit the definitions of interfaces of the previous section in order to group together functions (atomic components) that share interfaces of complexity higher than a given threshold. Two notions of interface complexity are implicitly considered here: popularity and collaboration.

4.1 Callee Popularity

If a function f is called by many different caller functions, then it is likely that any specific calling context in which f is called is not particularly important. For instance, functions used to store or access values in data structures like lists or hash tables are likely to be called by several different functions. In contrast, functions performing sub-tasks specific to a given function may be called only once or only by that function. In the latter case, the caller and the callee are more likely to obey tacit rules when communicating with each other, and breaking the interface between them by placing these functions in different units may be risky.

In what follows, let the *popularity* of a function f be the number of functions g that call f. (The popularity of a function could also be defined as the number of syntactic calls to that function; we consider here the former, simpler definition in order to determine if it is already sufficient to obtain interesting results.)

Formalizing the above intuition, our first algorithm generates a partition of units in which functions f and g are more likely to be assigned to the same unit if f calls g and g is not very popular. Given the set of control interfaces of a set S of functions, the algorithm starts by invoking the *PopularityGraph* subroutine to compute a weighted directed graph $G = (V, E)$ with the set of nodes $V = S$, where the directed edges in E denote calls between functions in S, and each edge weight is set to the popularity of the callee (i.e., the popularity of the function corresponding to the destination node of the edge).

Next, as long as the popularity graph $G = (V, E)$ is not empty, the algorithm proceeds with the following steps. Given a *cutoff policy*, it chooses a maximum popularity cutoff c by invoking the subroutine *ChooseCPCutoff*, and (temporarily) removes edges above that threshold by calling the subroutine *FilterPopularEdges*. The resulting subgraph G' is traversed top-down (or top-down in some spanning tree in case the subgraph is strongly connected). This traversal is performed by repeatedly invoking the subroutine *TopLevel* which returns a *top level node* in $G' = (V', E')$: a node v in V' is said to be a top level node if no other node u exists in V' such that $(u, v) \in E'$. Note that a (nonempty) directed graph has no top level node if and only if every node is part of some strongly connected component; in that case the subroutine *TopLevel* returns an arbitrary $v \in V'$. For each top level node t, the set of reachable nodes in G' from t is computed by invoking the subroutine *Reachable*. If this set of nodes is non-trivial (i.e., larger than one node), this set of nodes is defined as a unit. Nodes that would form trivial units (consisting of only themselves) are not allocated at this stage, but will be allocated at some subsequent iteration of the outer **while** loop with a possibly different (higher) popularity cutoff. Nodes corresponding to functions allocated to units are removed from G and from the subgraph G' using the *RemoveNodes* subroutine. When the inner **while** loop exits, a new iteration of the outer **while** loop may begin, and the entire process described above is repeated until all the functions have been allocated to some unit, at which point the outer **while** loop exits, and the algorithm terminates. At that point, every function in S has been allocated to exactly one unit in the resulting set U of units.

Algorithm 1 uses the following subroutines:

1. The subroutine *PopularityGraph* takes as input a set of (boolean) control interfaces of a set of functions S, and returns a weighted directed graph $G = (V, E)$. The graph is such that there is a vertex $v_f \in V$ for each function $f \in S$, and there is an edge $(v_f, v_g) \in E$ with weight w for each call from f to g in S, where w is the popularity of g.

2. The subroutine *IsEmpty* takes as input a graph $G = (V, E)$ and returns $True$ if $V = \emptyset$, and $False$ otherwise.

3. The subroutine *ChooseCPCutoff* takes as input a weighted directed graph G and returns a value c based on the *cutoff policy*. A cutoff policy is an external parameter to the algorithm. We considered and experimented with two types of cutoff policies in conjunction with this algorithm: policy `cp`n makes *ChooseCPCutoff* return the value n on the first invocation and the maximum weight in G on subsequent invocations, while policy `cpi` makes *ChooseCPCutoff* return the smallest weight in G.

4. The subroutine *FilterPopularEdges* takes as inputs a weighted directed graph $G = (V, E)$ and a value c. It returns a weighted directed graph $G' = (V, E')$ such that $E' \subseteq E$, and for all $e \in E$ of weight w, we have $e \in E'$ if and only if $w \leq c$.

5. The subroutine *TopLevel* takes as input a (nonempty) weighted directed graph $G' = (V', E')$ and returns any node v' such that there is no $v \in V'$ such that $(v, v') \in E'$. If such a node v' does not exist (i.e., every node in G' is part of some strongly connected component), then *TopLevel*(G') returns any $v' \in V'$.

6. The subroutine *Reachable* takes as inputs a graph $G = (V, E)$ and a node $t \in V$, and returns the set of nodes $V' \subseteq V$ reachable from t in G.

7. The subroutine *RemoveNodes* takes a graph $G = (V, E)$ and a set $u \subseteq V$ and returns a graph $G' = (V', E')$ such that $V' = V \setminus u$, and (V', E') is the subgraph of G induced by V'.

THEOREM 1. *For a call graph $\mathcal{G}$ over a set of functions S and given a set of control interfaces of S, the algorithm* $\mathsf{PartitionCP}(S)$ *creates a partition of $\mathcal{G}$.*

PROOF. We prove (i) that the algorithm terminates, and (ii) that when it has terminated, (a) every function in S is allocated to some unit u generated by the algorithm, and (b) that no function $f \in S$ is allocated to two units u and u' generated by it.

The subroutine *ChooseCPCutoff*, for all cutoff policies, eventually returns the maximum weight in G. Thus, the condition $G = G'$ in the **if** condition in the inner **while** loop is satisfied eventually. From that point onwards, at least one vertex is removed from the graphs G and G' in each iteration of the inner **while** loop. Since the graph G' is finite, the inner **while** loop must eventually exit when G' becomes empty. Also, since the graph G is finite, the outer **while** loop must eventually exit when G becomes empty. Thus, the algorithm terminates.

Since every function in S is a vertex in the popularity graph G, and since the outer loop runs until the graph G is empty (has no more vertices), and since a vertex is removed from G only if it is allocated to a generated unit, it follows that every function in S is allocated to some unit. For the part (ii)(b), we observe that, when a function f is allocated to a unit u (in line 9 of the algorithm), it is next removed from both G and G', and thus cannot be allocated to more than one unit. $\square$

4.2 Shared Code

If two functions f and g call many of the same functions, then it is likely that the higher-level operations they perform are functionally more related than with other functions that call a completely disjoint set of sub-functions. Therefore, functions that share a lot of sub-functions should perhaps be grouped in a same unit.

This intuition is formalized by our second partitioning algorithm, which generates a partition of units in which functions f and g are more likely to be assigned to the same unit if they have a relatively high degree of *collaboration*, where the degree of *collaboration* is the number of common functions called by both f and g.

Given a set of control interfaces of a set S of functions, the algorithm first creates a weighted undirected *collaboration graph* $W = (V, E)$ with $V = S$, and $E = (S \times S) \setminus \{(f, f) \mid f \in S\}$ (intuitively, there is an edge between any two distinct functions), and each edge weight is set to the number of sub-functions shared between the two functions the edge connect. Since an approach based on a single cutoff classifying inter-function collaboration into a boolean "high" or "low" is too coarse-grained, we instead propose a more general algorithm based on multiple collaboration thresholds. Given a *collaboration classification* policy (embodied

Algorithm 2 PartitionSC(S)

Input: A set of control interfaces of a set S of functions
Output: A partition of S into a set U of units
Variables: *WeightedGraph W*
1: $W :=$ *CollaborationGraph*(S)
2: $c :=$ *NumberOfCollaborationClasses*(G)
3: $L :=$ *CollaborationThresholds*(G, c)
4: **while** ($\neg$ *IsEmpty*(W)) **do**
5: $t := \max(L)$
6: $W' :=$ *FilterLightEdges*(W, t)
7: **while** ($\neg$*IsEmpty*(W')) **do**
8: $u :=$ *ConnectedComponent*(W')
9: **if** ($|u| > 1$ **or** $W' = W$) **then**
10: **add** u as a new unit in U
11: $W :=$ *RemoveNodes*(W, u)
end if
12: $W' :=$ *RemoveNodes*(W', u)
end while
13: $L := L \setminus \{t\}$
end while

by the *NumberOfCollaborationClasses* subroutine) denoting the number c of collaboration classes, the algorithm invokes the *CollaborationThresholds* subroutine to compute a set of c *collaboration thresholds* representing a sequence of *minimum collaboration levels* the algorithm will run through in descending order in subsequent iterations of the outer **while** loop, and edges representing collaboration levels below the minimum currently chosen will be (temporarily) removed by the *FilterLightEdges* subroutine invoked soon afterwards in the outer **while** loop. The outer **while** loop runs as long as the current collaboration graph W is not empty. At the beginning of each iteration, the maximum value t in the current list L of collaboration thresholds is found, and passed to the *FilterLightEdges* subroutine which returns a subgraph W' of the collaboration graph W in which only edges with weights higher than or equal to t remain. As long as the subgraph W' is not empty, the inner **while** loop runs. It invokes the *ConnectedComponent* subroutine on W' to find a group of nodes u in W' that are all reachable from each other. If a group of cardinality greater than 1 is found, or if no edges had been filtered out of W to get W' in this current iteration, the newly discovered group of nodes u is allocated as a new unit, and the nodes in u are removed from both W and W' using the *RemoveNodes* subroutine. Otherwise, the nodes in u are not yet allocated as an unit, and are removed from W' but not from W; indeed, nodes in u will be allocated during subsequent iterations of the outer **while** loop with lower values of the collaboration threshold. When W' becomes empty, the inner **while** loop terminates, the current collaboration threshold t that was used is discarded from L, and the next iteration of the outer **while** loop continues with the next lower value in L as the new current collaboration threshold. Eventually, when W is empty, the outer **while** loop terminates, and the algorithm terminates. At that point, every function in S has been allocated to exactly one unit in the resulting set U of units.

Algorithm 2 uses the following subroutines:

1. The subroutine *CollaborationGraph* takes as input a set of (boolean) control interfaces of a set of functions S and creates a weighted undirected graph $W = (V, E)$. The graph is such that there is a vertex $v_f \in V$ for each function f in S, and an edge $(v_f, v_g) \in E$ of weight w for every pair of functions f and g in S such that w is the degree of collaboration between f and g.

2. The subroutine *NumberOfCollaborationClasses* takes as input a collaboration graph and returns a positive integer c based on the *collaboration classification policy*; the policy `sc`n forces the subroutine to always return the value n.

3. The subroutine *CollaborationThresholds* takes as inputs a collaboration graph W and an integer c, and returns a sequence of c distinct non-negative integers starting from 0 and dividing equally the interval between 0 and the maximum weight in W.

4. The subroutine *FilterLightEdges* takes a weighted undirected graph $W = (V, E)$ and returns a graph $W' = (V, E')$ such that $E' \subseteq E$, and for all edges $e \in E$ of weight w we have $e \in E'$ if and only if $w \geq c$.

5. The subroutines *IsEmpty* and *RemoveNodes* are defined as before.

6. The subroutine *ConnectedComponent* takes as input a weighted undirected graph $W' = (V, E)$ and returns a set $u \subseteq V$ of nodes. The set u is such that every pair of nodes $v_1, v_2 \in u$ are connected in W', and for all nodes $v_3 \in V$, if $v_3 \notin u$, then for all $v_1 \in u$, v_1 and v_3 are not connected.

THEOREM 2. *For a call graph $\mathcal{G}$ over a set of functions S and given a set of control interfaces of S, the algorithm* PartitionSC(S) *creates a partition of $\mathcal{G}$.*

PROOF. We prove (i) that the algorithm terminates, and (ii) that when it has terminated, (a) every function in S is allocated to some unit u generated by the algorithm, and (b) that no function $f \in S$ is allocated to two units u and u' generated by it.

The subroutine *CollaborationThresholds* returns a list of integers in which the last element is the maximum weight in the collaboration graph W. Since one element from that list is discarded when the inner **while** loop exits, and since the inner **while** loop is guaranteed to exit (since it removes at least one vertex from the finite graph W' in each iteration and exits when W' has no more vertices), the last element of the list L is eventually assigned to t (in line 5). Thus, the condition $W' = W$ in the **if** condition in the inner **while** loop is satisfied eventually. From that point onwards, at least one vertex is removed from the graphs W and W' in each iteration of the inner **while** loop. Since the graph W is finite, the outer **while** loop must eventually exit when W becomes empty. Thus, the algorithm terminates.

Since every function in S is a vertex in the collaboration graph W, and since the outer loop runs until the graph W is empty, and since a vertex is removed form W only if it is allocated to a generated unit, it follows that every function in S is allocated to some unit. For the part (ii)(b), whenever a function f is allocated to a unit u (in line 10 of the algorithm), it is next removed from both W and W', and thus cannot be reallocated later to another unit. $\square$

Figure 1 shows a small fragment of the `osip` call-graph, chosen for simplicity. The shaded boxes show the partition

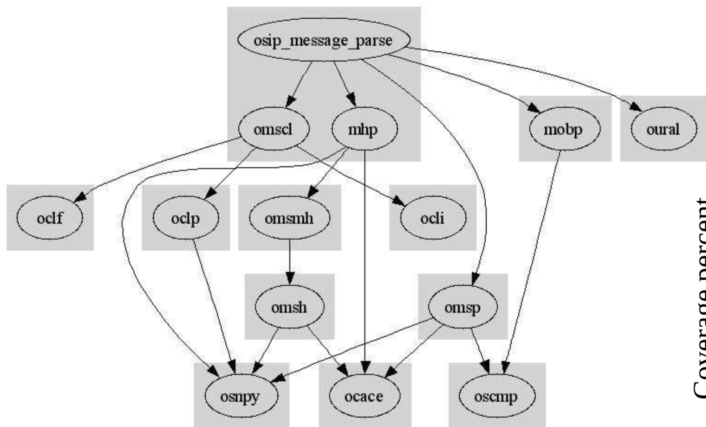

Figure 1: The call-graph, and the partition created by `sc7`

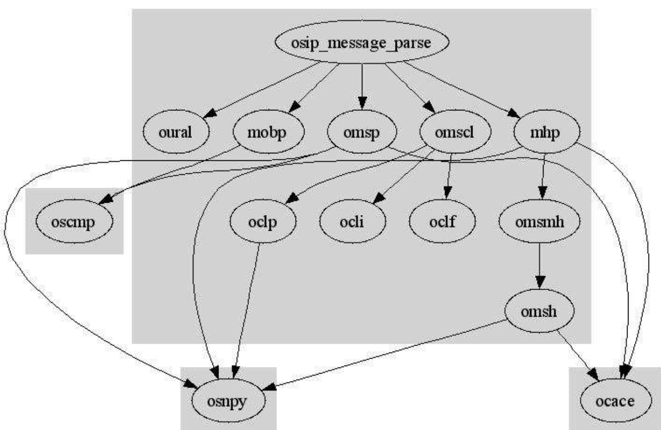

Figure 2: The partition created by `cp1i`

created by running `sc7` on the graph. On the same graph, running `cp1i` partitions the functions as shown by Figure 2. Note that each of the functions in the big fragment have 1 or 0 callers, whereas `oscmp`, `osnpy`, and `ocace` are more generic, having multiple callers. Running `cp3` partitions the functions into two groups: {`osnpy`} and the rest; `osnpy` is the only function with more than three callers.

5. EXPERIMENTAL RESULTS

We have implemented the algorithms presented in the previous section in a prototype tool for partitioning programs written in C. In order to evaluate the effectiveness of these algorithms, we applied our tool to a large[1] software application: oSIP, an open-source implementation of the *Session Initiation Protocol*. SIP is a telephony protocol for call-establishment of multi-media sessions over IP networks (including "Voice-over-IP"). oSIP is a C library available at `http://www.gnu.org/software/osip/osip.html`. The oSIP library (version 2.2.1) consists of about 30,000 lines of C code. Two typical applications are SIP clients (such as

[1]From a unit testing and verification point of view.

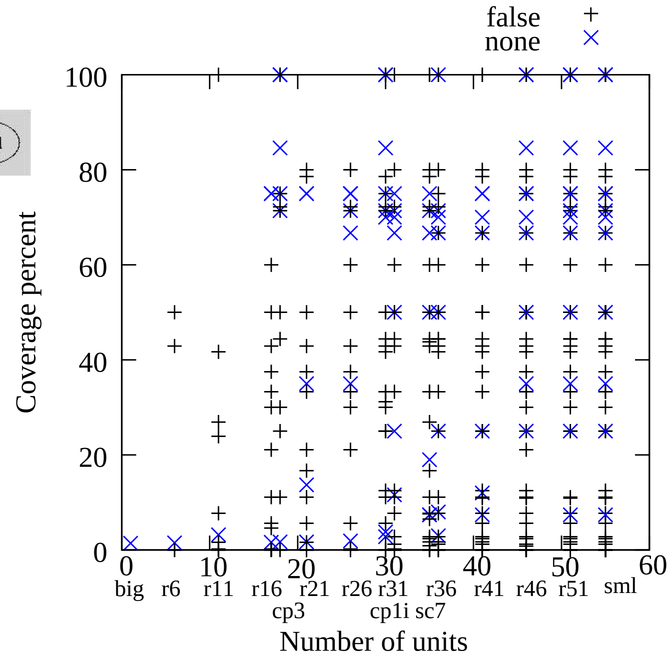

Figure 3: Coverage and incidences of false alarms

softphones to make calls over the internet from a PC) and servers (to route internet calls). SIP messages are transmitted as ASCII strings and a large part of the oSIP code is dedicated to parsing SIP messages.

Our experimental setup was as follows. We considered as our program P a part of the oSIP parser consisting of the function `osip_message_parse` and of all the other oSIP functions called (directly or indirectly) by it. The resulting program consists of 55 functions, which are described by several thousands lines of C code.[2] The call graph of this program is actually acyclic, which we believe is fairly common. (Note that our partitioning algorithms are not customized to take advantage of this property.) It is worth observing that the "popularity" (as defined earlier) distribution in this program is far from uniform: about half the functions (25 out of 55) are very unpopular, being called by only 1 or 2 callers, about 20 have around 5 callers each, and the remaining 8 are very popular, with 20 or more callers.

We used an extension of the DART implementation described in [17] to perform all our testing experiments. For each unit, the inputs controlled by DART were the arguments of the unique toplevel function of that unit, and DART was limited to running a maximum of 1,000 executions (tests) for testing each unit. In other words, if DART did not find any error within 1,000 runs while testing a unit, testing would then move on to the next unit, and so on. Since the oSIP code does not contain assertions, the search performed by DART was limited to finding segmentation faults (crashes). Whenever DART triggered such an error, the testing of the corresponding unit was stopped. Because of the large number of errors generated by all these experiments (see below), we could not visually inspect each of those; we therefore assumed that the overall program could not crash on any of its inputs and hence that all the errors

[2]The C files containing all these functions represent a total of 10,500 lines of code.

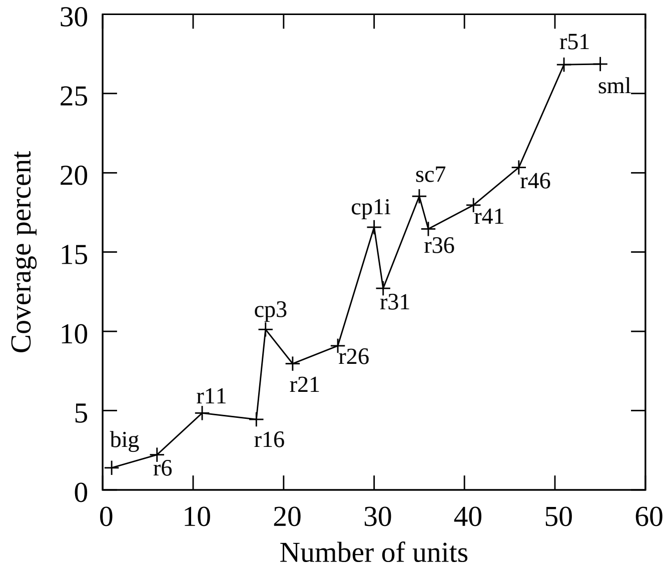

Figure 4: Overall branch coverage

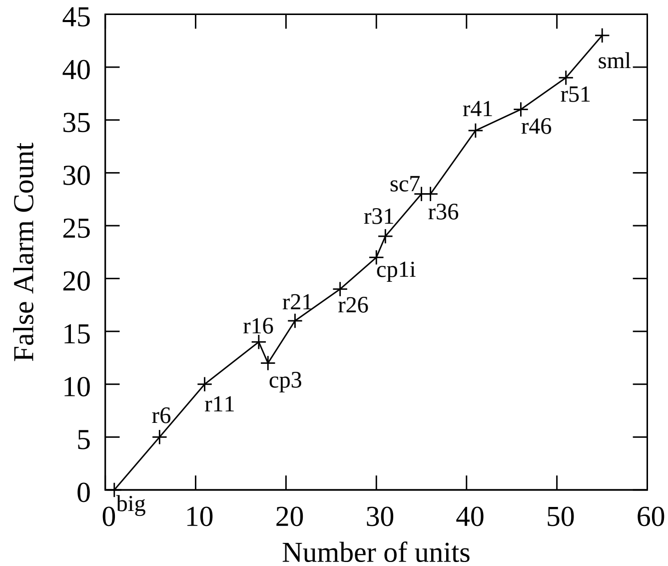

Figure 5: Number of false alarms

found by DART were spurious, i.e., false alarms. Thus, in this experimental setup, at most one false alarm can be reported per unit. Coverage is defined as branch coverage. Test coverage while testing a specific unit is defined and measured with respect to the code (branches) of that unit only.

We performed experiments with partitions generated by several partitioning algorithms: the symbol `cp1i` denotes the partition generated by *both* the partitioning algorithm that uses "callee popularity" iteratively *or* with a cutoff of 1, as both partitioning algorithms happened to generate the same partition for the oSIP code considered here; `cp3` represents the partition generated by "callee popularity" with a cutoff of 3; and `sc7` denotes the partition generated by the "shared code" partitioning algorithm with a policy value of 7. The above parameter values were chosen arbitrarily. To calibrate our results, we also performed experiments with the extreme partition consisting of a single unit containing all the functions, denoted by `big`, and with the other extreme partition where each unit consists of a single function, denoted by `sml`. Finally, we also performed experiments with a set of randomly generated partitions, each denoted by rn where n is the number of units of the corresponding partition.

Our experimental results are shown in a series of figures. Figure 3 shows for each partition, the coverage obtained for each unit in the partition, and whether a false alarm was reported for that unit. A (blue) $\times$ mark indicates no false alarm was reported (i.e., the unit could be successfully tested 1,000 times with different inputs), while a (black) + mark indicates that DART reported a false alarm (i.e., found a way to crash the unit within a 1,000 runs). (Thus, for n units, there are n marks on the corresponding column, but some of these are superposed and not distinguishable.) All those experiments took about one day of runtime on a Pentium III 800Mhz processor running Linux. Note the low coverage of 1% in `big`; this indicates that monolithic testing is severely ineffective. Also from this figure (and subsequent figures), it can be seen that the increase in coverage in the units of partitions to the right (which contain more units) is not free of cost since it also yields more false alarms.

For each partition considered, Figure 4 shows the overall branch coverage obtained after testing all the units in the corresponding partition. As is clear from Figure 4, the branch coverage in a partition tends to increase as the number of units in the partition rises and the average size of each unit in the partition gets smaller. At the extremes, coverage rises from about 1% in `big` to about 27% in `sml`.

These perhaps seemingly low coverage numbers can be explained as follows: the total number of program branches is large (1,162), not all the branches are executable in general or in our specific experimental setup –for instance, we do not inject errors for calls to `malloc`, `free`, etc. so branches that are executed in case of errors to calls to these functions are not executable–, testing of a unit stops as soon as an error is found or after 1,000 tests have been run, and finally, whenever DART cannot reason symbolically about some input constraint, it reduces to random testing [17] and random testing is known to usually yield very low code coverage [32, 17].

Note that the increase from `big` to `sml` is not monotonic: there are peaks corresponding to each of our partitioning algorithms `cp3`, `cp1i` and `sc7`. Thus, even though overall coverage rises as the number of units in the partition increases, our partitioning algorithms are able to select the units cleverly enough to raise the inter-unit cohesion sufficiently to consistently beat the overall coverage obtained by random partitions with similar numbers of units.

From Figure 4 it is clear that coverage on the whole rises as the number of units in a partition increases, and that `sml` is the best partition with respect to coverage. However, this higher coverage is obtained at the cost of an increased number of false alarms, as can be seen from Figure 5, which gives the absolute number of false alarms found for each partition.

We thus have a tension between two competing objectives: maximizing overall test coverage while minimizing the number of false alarms. In order to evaluate how the different partitioning algorithms satisfy these two competing

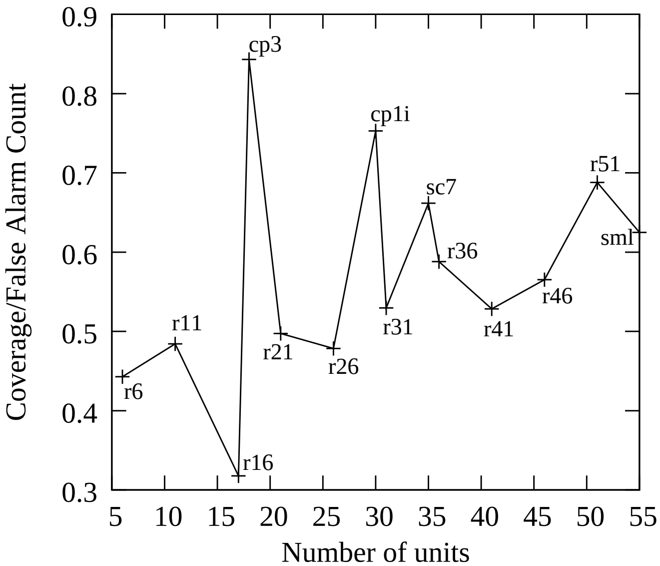

Figure 6: Ratio of coverage to false alarm

objectives, Figure 6 shows the ratio of the overall coverage obtained with a partition divided by the number of false alarms reported for the partition. (The value of this ratio is undefined for `big` as it has zero false alarms; however, `big` is not a serious contender for the title of best partitioning scheme as it leads to very low code coverage.) Observe that `cp3`, `cp1i`, and `sc7` each correspond to clear peaks in the graph, although of steadily decreasing magnitude, indicating that `cp3` is the best suited partitioning scheme among those considered for the specific code under analysis. These peaks mean that all three algorithms clearly beat neighboring random partitions, even though these three algorithms exploit only simple control interface definitions ignoring all data dependencies. Whether using more elaborate interface definitions like those discussed in Section 3 can significantly improve those results is left for future work.

6. DISCUSSION AND OTHER RELATED WORK

Interfaces are defined in the previous sections as *syntactic* notions, which are computable via static analysis, and the complexity of interfaces is then used as a *heuristic* measure of how intertwined functions are. However, an ideal software partitioning algorithm would cut apart a function f calling another function g only if any input of g is *valid*, i.e., does not trigger a false alarm. For instance, if function g takes as input a boolean value, it is very likely that both 0 and 1 are valid inputs. In contrast, if g takes as argument a pointer or a complex data structure, its inputs are likely to be *constrained* (e.g., g may assume that an input pointer is non-null, or that the elements of its input array are sorted): g assumes that all its inputs satisfy a *precondition*.

Unfortunately, inferring preconditions automatically from program analysis (dubbed the *constraint inference problem* in [16]) is obviously as hard as program verification itself. Another strategy would be to perform piecemeal testing (see Section 2), then hide interfaces where false alarms were generated, and repeat the process until all false alarms have been eliminated. This strategy looks problematic in practice because it would require the user to examine all the errors generated to determine whether they are spurious or not, the number of iterations before reaching the optimum could be large, and the partition resulting from this process may often end up being the entire system, that is, no partition at all. Another impractical solution would be to generate and evaluate all possible partitions to determine which one is best. All these practical considerations explain why we seek in this work *easily-computable syntactic heuristics* that can generate an *almost optimal* partition at a *cheap cost*.

In spirit, our work is closest to work on automatically decomposing hardware combinatorial and sequential circuits into smaller pieces for compositional circuit synthesis, optimization, layout and automatic test-pattern generation (e.g., [21, 12, 33]). Circuit topology has also been exploited for defining heuristics for BDD variable orderings and for finding upper bounds on the sizes of BDD representations of combinatorial circuits (e.g., [4, 28]). However, the components, interfaces and clustering algorithms used for hardware decomposition are fairly different from those discussed in this paper. Indeed, a digital hardware component is always finite state and its interface to the rest of the world is simply a set of pins (booleans). In contrast, defining exactly what the interface of a software component (say a C program) is already challenging (the flow of control between functions can be hard to determine, functions may take unbounded data structures as inputs, side-effects through global variables are possible and sometimes hard to predict, etc.), and program verification for Turing-expressive languages is in general undecidable due to their possibly infinite state spaces.

Our work is also related to metrics for defining software complexity, such as *function points* (e.g., [14]) among others, and to work on *software architecture reconstruction* (e.g., [35]) that aim to facilitate the understanding of legacy code for reuse and refactoring purposes. Work on code reuse usually consider more sophisticated notions of "components" and "interfaces", which are often obtained through a combination of static, dynamic and manual analyses, but do not attempt to automatically partition code for facilitating a subsequent more precise automated program analysis.

Software partitioning for effective unit testing is related to compositional verification, which has been extensively discussed in the context of the verification of concurrent reactive systems in particular (e.g., see [18, 7, 1]). Compositional verification requires the user to identify components that can be verified in isolation and whose abstractions are then checked together. To the best of our knowledge, we are not aware of any work on heuristics for automatic software partitioning with the goal of compositional verification. The software partitioning heuristics presented in this work could also be used for suggesting good candidates of units suitable for compositional verification. However, it is worth emphasizing that the focus of this work has been (so far) the decomposition of *sequential* programs described by a set of functions, whose behaviors are typically more amenable to compositional reasoning than those of concurrent reactive systems, where compositional analysis (testing or verification) is arguably more challenging.

Algorithms for inter-procedural static analysis (e.g., [24, 9, 20]) and pushdown model checking (e.g., [8, 2]) are also compositional, in the sense that they can be viewed as analyzing individual functions in isolation, summarizing the results of these analyses, and then using those summaries to perform a global analysis across function boundaries. In contrast, the software partitioning techniques we describe in this paper are more light-weight since they provide only heuristics based on an analysis of function interfaces only (not the full function code), and since no summarization of unit testing nor any global analysis is performed. Software partitioning could actually be used to first decompose a very large program into units, which could then be analyzed individually using a more precise inter-procedural static analysis (since a single unit may contain more than one function). However, evaluating the effectiveness of a partitioning scheme when used in conjunction with static analysis tools (including static software model checkers like SLAM [3] or BLAST [23], for instance) would have to be done differently than in this paper since (1) static analysis usually performs a for-all path analysis and hence does not measure code coverage as is done during testing, and (2) static analysis reports (typically many) false alarms due to abstraction and the imprecision that it always introduces, in addition to false alarms simply due to missing environment assumptions as with testing. Another interesting problem for future work (first suggested in [16]) is how to perform automatic dynamic test generation compositionally using a summarization process similar to what is done in inter-procedural static analysis.

Finally note that, although we used a DART implementation to perform the experiments reported in the previous section, the software partitioning problem and the techniques that we proposed to address it are independent of any particular automated test generation framework. We refer the reader to [17, 16] for a detailed discussion on other automated test generation techniques and tools.

7. CONCLUSION

We studied in this paper how to automatically partition a large software program into smaller units that can be tested in isolation using automated test generation techniques without generating (too many) false alarms due to unrealistic inputs being injected at unit interfaces exposed by the partitioning. We proposed an approach that identifies control and data inter-dependencies between program functions using static analysis, and divides the source code into units where highly-intertwined functions are grouped together. We presented several partitioning algorithms based on this idea.

Preliminary experiments show that, perhaps surprisingly, *even partitioning algorithms exploiting only simple control dependencies can already significantly increase test coverage without generating too many false alarms*. These experiments also seem to validate the intuition behind these algorithms, namely that *grouping together highly-intertwined functions in the same unit improves the effectiveness of testing*, since those algorithms are able to consistently beat random partitions with similar numbers of units for our benchmark.

More experiments are needed to confirm these observations. Also we do not claim that our specific partitioning algorithms are the best possible: we have only shown that *there exist some simple partitioning algorithms that can beat random partitions*, but other partitioning algorithms (and parameter values) should be experimented with. Whether using more elaborate interface definitions like those discussed in Section 3 can improve those results is also left to be investigated.

Still, we believe our preliminary results are an encouraging *first step towards defining light-weight heuristics to partition large software applications* into smaller units that are amenable to (otherwise *intractable*) more precise analyses, such as dynamic software model checking.

Acknowledgements

We thank Nils Klarlund for helpful comments on preliminary ideas that led to this work. This work was funded in part by NSF CCR-0341658.

8. REFERENCES

[1] L. Alfaro and T. A. Henzinger. Interface Theories for Component-Based Design. In *Proceedings of the 1st International Workshop on Embedded Software (EMSOFT)*, pages 148–165. Springer-Verlag, Tahoe City, October 2001.

[2] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis. Analysis of Recursive State Machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(4):786–818, July 2005.

[3] T. Ball and S. Rajamani. The SLAM Toolkit. In *Proceedings of the 13th Conference on Computer Aided Verification (CAV)*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264, Springer-Verlag, Paris, July 2001.

[4] C. L. Berman. Circuit Width, Register Allocation and Ordered Binary Decision Diagrams. *IEEE Transactions on Computer-Aided Design*, 10(8):1059–1066, August 1991.

[5] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating Test from Counterexamples. In *Proceedings of the 26th International Conference on Software Engineering (ICSE)*. ACM Press, Edinburgh, May 2004.

[6] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133, ACM Press, Rome, July 2002.

[7] T. Bultan, J. Fischer, and R. Gerber. Compositional Verification by Model Checking for Counter-Examples. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 224–238. ACM Press, San Diego, January 1996.

[8] O. Burkart and B. Steffen. Model Checking for Context-Free Processes. In *Proceedings of the 3rd International Conference on Concurrency Theory (CONCUR)*, pages 123–137, Springer-Verlag, Stony Brook, August 1992.

[9] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 30(7):775–802, 2000.

[10] C. Cadar and D. Engler. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *Proceedings of the 12th International SPIN Workshop*

on Model Checking of Software (SPIN), volume 3639 of *Lecture Notes in Computer Science*, pages 2–23, Springer-Verlag. San Francisco, August 2005.

[11] C. Csallner and Y. Smaragdakis. Check'n Crash: Combining Static Checking and Testing. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 422–431, ACM Press, St. Louis, May 2005.

[12] S. R. Das, W.-.-B. Jone, A. R. Nayak, and I. Choi. On Testing of Sequential Machines Using Circuit Decomposition and Stochastic Modeling. *IEEE Transactions on Systems, Man, and Cybernetics*, 25(3):489–504, March 1995.

[13] J. Edvardsson. A Survey on Automatic Test Data Generation. In *Proceedings of the 2nd Conference on Computer Science and Engineering*, pages 21–28, Linköping, October 1999.

[14] S. Furey. Why We Should Use Function Points. *IEEE Software*, 14(2), 1997.

[15] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL)*, pages 174–186, ACM Press, Paris, January 1997.

[16] P. Godefroid and N. Klarlund. Software Model Checking: Searching for Computations in the Abstract or the Concrete (Invited Paper). In *Proceedings of 5th International Conference on Integrated Formal Methods (IFM)*, volume 3771 of *Lecture Notes in Computer Science*, pages 20–32, Springer-Verlag, Eindhoven, November 2005.

[17] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223, ACM Press, Chicago, June 2005.

[18] O. Grumberg and D. E. Long. Model Checking and Modular Verification. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):843–871, 1994.

[19] N. Gupta, A. P. Mathur, and M. L. Soffa. Generating test data for branch coverage. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE)*, pages 219–227, IEEE Computer Society Press, Grenoble, September 2000.

[20] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A System and Language for Building System-Specific Static Analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, pages 69–82, ACM Press, Berlin, June 2002.

[21] S. Hassoun and C. McCreary. Regularity Extraction via Clan-Based Structural Circuit Decomposition. In *Proceedings of the 1999 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 414–419, ACM Press, San Jose, November 1999.

[22] R. Hastings and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 125–138, Berkeley, January 1992.

[23] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL)*, pages 58–70, ACM Press, Portland, January 2002.

[24] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 104–115, ACM Press, New York, October 1995.

[25] J. C. King. Symbolic Execution and Program Testing. *Journal of the ACM*, 19(7):385–394, 1976.

[26] B. Korel. A dynamic Approach of Test Data Generation. In *IEEE Conference on Software Maintenance*, pages 311–317, IEEE Computer Society Press, San Diego, November 1990.

[27] E. Larson and T. Austin. High Coverage Detection of Input-Related Security Faults. In *Proceedings of 12th USENIX Security Symposium*, Washington D.C., August 2003.

[28] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[29] G. J. Myers. *The Art of Software Testing*. Wiley, 1979.

[30] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL)*, pages 128–139, ACM Press, Portland, January 2002.

[31] The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology, Planning Report 02-3, May 2002.

[32] J. Offutt and J. Hayes. A Semantic Model of Program Faults. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 195–200, ACM Press, San Diego, January 1996.

[33] M. R. Prasad, P. Chong, and K. Keutzer. Why is ATPG easy? In *Proceedings of the 36th Design Automation Conference (DAC)*, pages 22–28, ACM Press, New Orleans, June 1999.

[34] Valgrind. web page: `http://valgrind.org/`.

[35] A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva. Symphony: View-Driven Software Architecture Reconstruction. In *Proceedings of the 4th IEEE/IFIP Working Conference on Software Architecture (WICSA)*. IEEE Computer Society Press, Oslo, June 2004.

[36] W. Visser, C. Pasareanu, and S. Khurshid. Test Input Generation with Java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, ACM Press, Boston, July 2004.