Modeling Software Execution Environment

Dawei Qi[†] William N. Sumner[‡] Feng Qin[*] Mai Zheng[*] Xiangyu Zhang[‡] Abhik Roychoudhury[†]
[†]Dept. of Computer Science, National University of Singapore
[‡]Dept. of Computer Science, Purdue University
[*]Dept. of Computer Science and Engineering, The Ohio State University

***Abstract*—**
Software execution environment, interfaced with software through library functions and system calls, constitutes an important aspect of the software's semantics. Software analysis ought to take the execution environment into consideration. However, due to lack of source code and the inherent implementation complexity of these functions, it is quite difficult to co-analyze software and its environment. In this paper, we propose to extend program synthesis techniques to construct models for system and library functions. The technique samples the behavior of the original implementation of a function. The samples are used as the specification to synthesize the model, which is a C program. The generated model is iteratively refined. We have developed a prototype that can successfully construct models for a pool of system and library calls from their real world complex binary implementations. Moreover, our experiments have shown that the constructed models can improve dynamic test generation and failure tolerance.

I. Introduction

Software interacts with its environment through library functions and system calls. On one hand, part of software functionality is fulfilled by these functions; files and databases stored in external resources constitute an important part of program execution state. On the other hand, inputs from the environment, such as network messages, values from input devices, and signals indicating various exceptional states (e.g. file-system-full), affect software execution. As such, techniques that analyze programs should consider environment a cohesive part of program semantics and reason about behavior of a program in its proper environmental context.

Many program analysis, including both static and dynamic analysis, require proper reasoning about environment. For example, dynamic program slicing [1] and taint analysis [2] need to know the dependencies between input and output variables for a library/system function. A naïve approximation that assumes each output relies on all inputs leads to exponential growth in the size of slices or the number of tainted variables. Symbolic execution engines [3], [4] need to have appropriate models for library and system calls in order to construct correct symbolic constraints. Without such models, the underlying SMT solver may produce solutions that are infeasible at runtime. Additionally, program environment can make a program vulnerable to security attacks, e.g., giving away permissions to confidential files, as evidenced in a recent study [5]. In this scenario, neither the program nor the system call implementations are buggy on their own, whereas the combination of both leads to the vulnerabilities. Without a proper model of environment, an analysis cannot draw a definite conclusion about the safety of a program. Recently, techniques were proposed to survive software failures, especially for those long running critical server programs [6]. The effectiveness of such techniques hinges on precise environmental models. For example, one of the ideas is to enlarge an application's memory allocation request by padding additional bytes to survive buffer overflows. Without knowing the behavior of the underlying memory allocator, it is difficult to achieve effective and optimal padding.

Despite its importance, modeling environment is challenging. First, it is often difficult to acquire the source code of system and library functions. Without source code, it is hard to reason about the semantics of those functions. Moreover, even if source code is available, the code base is often prohibitively large and complex, making the modeling task substantially challenging. For example, Linux-2.6.38 contains 341 system calls with more than 14 millions lines of source code and glibc-2.8 provides 1234 functions with more than 1.3 millions lines of code. Also, library and system call implementations tend to be complex as they have gone through substantial optimizations in pursuit of performance.

Currently, environment models in most existing program analysis are manually constructed by either tool developers or users [3], [4]. Due to the scale and complexity of library and system call implementations, only a very small portion of all these calls are modeled, and only a single aspect, usually the most basic functional aspect, of a call is modeled. For example, a model for a file system read in [3], [4] only contains the logic for retrieving file content but lacks the logic for setting the error code. However, different analysis may be interested in different aspects of a library/system call. Furthermore, since these models are manually constructed, they lack flexibility. If a different version of a library or kernel is used, it is difficult to adjust the underlying models. For environment functions that are not well documented, it is almost impossible to construct their models manually.

Our goal is to *develop an automated technique that can construct models for library and system call functions from the executables of these functions*. Our models are essentially C programs that provide the same functionalities of the functions being modeled, yet substantially simplified. Such programs can be directly included as part of the application to enable program-environment co-analysis. We make the key observation that in many cases, the complexity of library and system functions is due to optimizations and the necessity of handling architecture dependent issues and exceptional signals. Most of this complexity can be suppressed in a high level behavioral model.

As the first step towards this goal, we show its feasibility by extending the oracle guided program synthesis technique [7] to model library and system functions. In particular, the original executable of a function is used as the oracle. By running the oracle with a set of inputs, we acquire a behavioral specification of the function, consisting of a set of inputs and the corresponding outputs. After this, a C program is synthesized to satisfy the specification via the following steps. From the description of a system/library function, we first speculate the possible data structures that can be used to construct its model. Then the primitive statements related to manipulating such data structures are provided as the building blocks of synthesis. Finally, the orderings of these building blocks are modeled as constraints, a solution to which produces a candidate program.

Our contributions are highlighted as follows.

- We propose a novel and practical application of program synthesis. It generates substantially simplified C code that models the behavior of system and library functions. It is one step towards automatic co-analysis of applications and their environment as the model code can be included as part of the user code and directly subject to various analysis.
- We identify the technical challenges and propose solutions. In particular, we identify a set of primitive components that are specific to the commonly used data structures for system and library functions, and develop their synthesis encodings. They are the fundamental building blocks of the models.
- We propose solutions to modeling loops. Expressive primitives are proposed to abstract the iterative semantics such that in many cases we can avoid synthesizing loops. For other cases, we formulate loops as high order functions that repeatedly apply a loop-free function to a sequence of inputs. The problem is hence reduced to modeling the loop-free function.
- To reduce the search space, we propose to explicitly encode type constraints so that we can avoid constructing models that are not well typed.
- We have implemented a prototype. It is able to model a pool of real world system and library calls, including those in file system. We also use the generated models in two analysis: dynamic test generation and failure tolerance. Our experiments have shown that the generated models can improve the analysis results.

II. Motivating Examples

In this section, we show two examples that motivate the importance of modeling environment.

A. Program Dependence Analysis

Program dependence analysis detects data and control dependences between statements. Being either static or dynamic, it is the core technique for program slicing [8], information flow tracking [9], and taint analysis [2]. Handling library and system calls is a prominent challenge in dependence analysis. First, the source code of these calls is often missing. Even if the source code is available, it is often too complex to analyze, due to the underlying optimizations, engineering tricks, and the correlations between functions. For example, many libc functions are optimized using bit operations, which require bit level dependence tracking. As a result, the dependences exercised in the bodies of these calls are invisible to the analysis. However, such dependences are important as statements in the user space may transitively depend on each other through these invisible dependences in the library/kernel space.

Consider the example in Fig. 1. The program first writes 40 values to the buffer `i_buf`. It then outputs the buffer through a file write service call (line 3). Later, it makes two `fgets()` calls at lines 5 and 9. The first invocation retrieves 20 bytes from the file and the second invocation retrieves the rest. Therefore, at line 10, the first byte in the buffer retrieved by the second invocation, `o_buf[0]`, is dependent on the definition of the 21st element of `i_buf` at line 3. Such dependence cannot be determined by most existing program dependence analysis engines without modeling file operations `fwrite()` and `fgets()`. Note that, a simple approach that assumes each output byte dependent on all input bytes would conclude that the use of `o_bug[0]` is dependent on the definitions of all the array elements, and even the file descriptor and the size (arguments of `fgets()`).

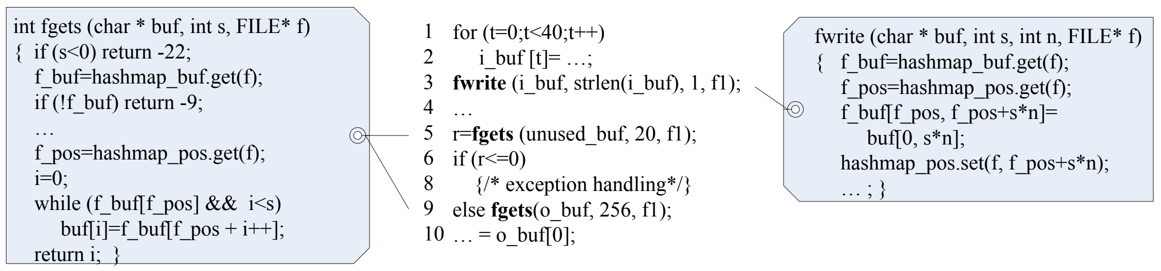
Moreover, if an error code is returned for the first `fgets()` invocation at line 5, the analysis cannot determine the precise dependence of the error code without the models. In contrast, from the desired model of `fgets()` shown in Fig. 1, it can be precisely determined that if the error code is -22, the dependence is on the size at line 5; if the error code is -9, the dependence is on the buffer pointer. Note that although the models are simple, the corresponding real implementations are substantially more complex, with the file buffer control and caching logic being the dominant factors of the code bodies.

B. Software Failure Tolerance

Software failure tolerance mechanisms are designed to survive software failures during production runs, which is very important for many applications (e.g., critical process control or on-line transaction monitoring) that demand high availability [10]. Rx [6] and FirstAid [11] are recent advances on surviving failures that are caused by deterministic bugs, such as buffer overflows and double frees. They do so by re-executing the failed program from previous checkpoints and exploiting execution environment changes.

Fig. 2 shows a buffer overflow bug in a stable version of Squid [12], which is a popular proxy server. In function `ftpBuildTitleUrl`, `t` is a heap buffer storing an input request. Line 4 calculates the expected buffer length `len`, which is incorrect since it does not consider special http characters [13]. Line 7 allocates the buffer based on `len`. As a result, buffer `t` can be overflowed at line 9 since `rfc1738_escape_part` may return a longer-than-expected string.

Both Rx and FirstAid attempt to survive the buffer overflow by using a fixed-size padding. They implicitly assume a simple

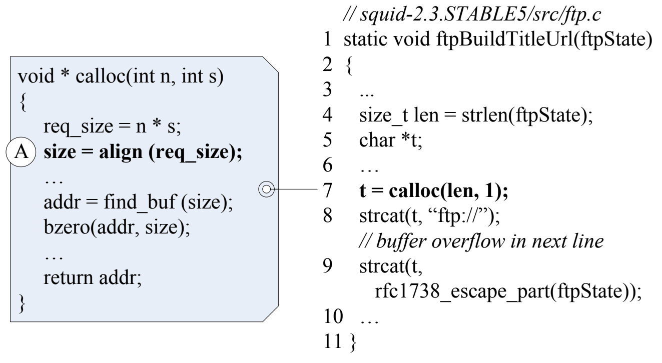


```
int fgets (char * buf, int s, FILE* f)
{  if (s<0) return -22;
   f_buf=hashmap_buf.get(f);
   if (!f_buf) return -9;
   …
   f_pos=hashmap_pos.get(f);
   i=0;
   while (f_buf[f_pos] &&  i<s)
       buf[i]=f_buf[f_pos + i++];
   return i;  }
```

```
for (t=0;t<40;t++)
    i_buf [t]= …;
fwrite (i_buf, strlen(i_buf), 1, f1);
…
r=fgets (unused_buf, 20, f1);
if (r<=0)
    {/* exception handling*/}
else fgets(o_buf, 256, f1);
… = o_buf[0];
```

```
fwrite (char * buf, int s, int n, FILE* f)
{   f_buf=hashmap_buf.get(f);
    f_pos=hashmap_pos.get(f);
    f_buf[f_pos, f_pos+s*n]=
        buf[0, s*n];
    hashmap_pos.set(f, f_pos+s*n);
    … ; }
```

Fig. 1. Models are needed for precise program dependence analysis. Shaded boxes are the desired models.



```
void * calloc(int n, int s)
{
    req_size = n * s;
(A) size = align (req_size);
    …
    addr = find_buf (size);
    bzero(addr, size);
    …
    return addr;
}
```

```
// squid-2.3.STABLE5/src/ftp.c
static void ftpBuildTitleUrl(ftpState)
{
  ...
  size_t len = strlen(ftpState);
  char *t;
  …
  t = calloc(len, 1);
  strcat(t, "ftp://");
  // buffer overflow in next line
  strcat(t,
     rfc1738_escape_part(ftpState));
  …
}
```

Fig. 2. Simplified code snippet from squid-2.3.STABLE5.

model of the memory management library, i.e., calloc adds no extra space to the requested buffer. This assumption may not hold. For example, glibc allocates memory blocks in the size of a multiple of eight (or sixteen on 64-bit systems) [14], as shown at (A) in the model in Fig. 2. Furthermore, some customized memory management libraries (e.g., [15], [16]) may append larger space to the user-requested memory for various reasons. In these scenarios, if not large enough, the padding added by Rx and FirstAid may be masked by the default padding from the underlying memory management library, rendering Rx and FirstAid ineffective. Additionally, the fact that padding cannot survive a failure may mislead developers in understanding the root causes since they may eliminate the possibility of a buffer overflow. Note while larger paddings increase the probability of surviving failures, they incur more resource consumption and/or larger runtime overhead. Similar situations apply to other environmental changes, such as determining the delay of recycling freed buffers for handling dangling pointers and double frees. Therefore, it is imperative to model the behavior of underlying memory management libraries to provide scientific basis for the effectiveness of Rx and FirstAid.

III. Overview and Technical Challenges

In this paper we focus on solving the following problem. *Given the executable of a library/system function, denoted as f_c, we aim to synthesize its model f_s, which is a simple program that has the same observable behavior as f_c.* We do not require the source code of f_c, but rather its binary form. Models are in C language so that they can be compiled and executed. They can also be included as part of the application to allow various application-environment co-analysis, without substantial extension of existing analysis.

TABLE I
Primitive components

Primitive	Explanation
input^n	the nth input.
const^c	represents constant c.
ge	ge(x1,x2) returns true is x1 is greater than or equal to x2, otherwise false
ite	ite(x1,x2,x3) returns x2 if x1 is true, otherwise x3
subarray	subarray(in_array_len,in_array,pos,len,out) returns the length of the output array, the length is "len" if $pos + len \leq in\_array\_len$, otherwise the length is $in\_array\_len - pos$. The output array is put in "out".
minus	minus(x1, x2) represents x1-x2

Given f_c, we first acquire an initial set of input and output pairs, each pair describing the observable output when executing f_c with an input. We then try to construct a model that manifests the same input/output behavior.

Consider the glibc file read function. It takes as inputs the file content, file position and the length of data to be read. Its outputs include the data and a return value. If the read operation succeeds, the return value is the length of the data that has been read. If it fails, the return value is -1. The signature of the function we expect to synthesize is as follows.

```
int read(int file_len, const char
      *file_content, int file_pos,
      int return_value, char *out_buf);
```

The first four arguments represent inputs and the last two represent outputs.

We extend the oracle-guided program synthesis technique [7] to model system and library calls. In particular, a set of primitive operations, or so called *components*, are provided. These components are the building blocks of models. The derivation of a model is essentially the process of searching for an order of these components such that the ordered sequence composes a program that manifests the specified input/output behavior. The primitive components include basic arithmetic operations, assignment statements, conditional statements, and more importantly, data structure operations. For example, the components for file read are presented and explained in Table I.

The search is conducted through an SMT solver. Specifically, variables are introduced to explicitly represent the location of each primitive component, called the *location variables*. The correlations between these location variables are encoded as constraints. For example, the location of a primitive that defines a variable x must precede the location of a primitive that uses x.

Consider the file read example. Assume a primitive component is the minus operation $z = y - x$. A variable l_z is introduced to represent the location of the primitive. The solver resolves $l_z = 9$, indicating the primitive should be put at line 9. The synthesized model for file read is shown in Fig. 3. Note that it precisely captures the behavior of file read with only 10 lines of code, whereas the real implementation of file read has over 700 lines of code, involving complex file buffer/cache control and memory management logic.

```
0:   o0 = input^0
1:   o1 = input^1
2:   o2 = input^2
3:   o3 = input^3
4:   o4 = const^(-1)
5:   o5 = ge(o4,o3)
6:   o6 = ite(o5,o4,o3)
7:   o7 = subarray(o0,o1,o2,o6,o8)
9:   o9 = minus(o0,o2)
     return o7 o8
```

Fig. 3. Synthesized program for "read", this synthesized program puts the right data into the read buffer if the parameters are valid. When the read length provided is a negative integer, it returns -1. Line 9 is dead code.

Technical Challenges. The existing oracle-guided program synthesis technique [7] is insufficient for modeling complex system and library calls. In particular, it handles very small code bodies. It only considers simple primitives such as arithmetic and bit operations as their goal was to synthesize computational cores, e.g. encoder/decoder, whereas the functionalities of system and library calls are mostly realized through data structures. It does not handle programs with loops. In this paper, we focus on solving the following technical challenges.

- System and library calls often have complex implementations, using a large volume of instructions with various kinds. On one hand, it is impossible and unnecessary to provide all such instructions as the initial primitives for model construction. On the other hand, it is insufficient to use only arithmetic and bit operations as primitive components. We need to identify the set of primitives that is specific for environment functions and develop their synthesis encodings.
- As the volume and complexity of the primitive components increases, the search space is enlarged. We need to develop techniques that help reduce the search space.
- Loops are heavily used in system and library calls. We need to develop solutions for loops.
- Although a constructed model must manifest the same observable behavior *regarding the given set of input and output pairs*. It remains uncertain if the generated model has the same behavior for *all inputs*. For system and library calls, it is unrealistic to assume the presence of a validator (a human expert or a theorem prover) that can prove the equivalence of the two. We have to develop solutions to mitigate such uncertainty.

IV. Notations

We use f_c and f_s to denote the original complex function and its simplified model, respectively. Both f_s and f_c have the same function signature. We denote the input variables and output variable for f_s/f_c as $\overrightarrow{I}$ and O respectively. Without loss of generality, each primitive component function is assumed to have only one output variable. We use $\overrightarrow{I_i}$ to represent the list of input variables for component g_i. The k^{th} input variable in $\overrightarrow{I_i}$ is denoted as $\overrightarrow{I_i}[k]$. We use O_i to represent the only output variable for g_i. The formal specification of g_i is denoted as $\phi_i(\overrightarrow{I}_i, O_i)$.

We use Q to represent all input variables for the primitive component functions, and we use R to represent all output variables for the primitive component functions.

$$Q = \bigcup_{i=1}^{N} \overrightarrow{I_i}, R = \bigcup_{i=1}^{N} \{O_i\}$$

For each variable x in $Q \cup R \cup \overrightarrow{I} \cup \{O\}$, we declare a location variable l_x. A location variable is interpreted in the following way: i) if $x \in R$, then the primitive whose output is x is placed at line l_x, and ii) if $x \in Q$, then x is the same as the output from line l_x, denoting x is used as input in a primitive and its definition is at l_x, and iii) if $x \in \overrightarrow{I}$, then output at line l_x is the same as input variable x of the synthesized function f_s, and iv) if $x == O$, then the output of f_s is the same as the output from line l_x. We use L to represent the set of all location variables.

We use E to represent all the input-output pairs that we have observed from the original function f_c. For each input-output pair (α_i, β_i) in E, $\beta_i == f_c(\alpha_i)$.

$$E = \{(\alpha_0, \beta_0), (\alpha_1, \beta_1), ..., (\alpha_m, \beta_m)\}$$

V. Basic Design

Similar to oracle-guided program synthesis [7], the basic design of our modeling technique is to encode the correlations between location variables and the input-output specification as constraints. A solution to these variables denotes a composition for the primitive components corresponding to f_s. This section describes the basic design. In the next two sections, we will describe our innovations over the basic design to handle practical challenges, i.e., identifying and encoding expressive primitives, providing the initial set of primitives, reducing search space, supporting loops, and validating models.

We first encode a type of constraints dictating the *well-formedness* of f_s. A well-formed program should satisfy the following constraints.

The inputs of any component function should always be defined before they are used, and there should be no two primitives placed on the same line,

$$\psi_1 = \bigwedge_{i=1}^{N} \bigwedge_{x \in \overrightarrow{I_i}, y \equiv O_i} l_x < l_y \ \wedge \bigwedge_{x,y \in R, x \not\equiv y} l_x \neq l_y$$

The first a few lines of the synthesized program are reserved for input variables. Therefore, all the location variables for inputs are fixed. All the other location variables should be within the correct range. Let $M = N + |\overrightarrow{I}|$, N being the number of primitives. The following two formula encode the constraints on the range of location variables.

$$\psi_2 = \bigwedge_{x \in Q} 0 \le l_x < M \wedge \bigwedge_{x \in R} |\overrightarrow{I}| \le l_x < M \wedge \bigwedge_{k=0}^{|\overrightarrow{I}|-1} l_{I[k]} = k$$

Constraint ϕ_{wfp} defines the well-formedness of a program.

$$\phi_{wfp} = \psi_1 \wedge \psi_2$$

The following constraint establishes the equality between variables across multiple components to denote data flow correlations.

$$\phi_{dataflow} = \bigwedge_{x,y \in Q \cup R \cup \overrightarrow{I} \cup \{O\}} l_x = l_y \Rightarrow x = y$$

For example, assume two primitives $o_1 = g_1(i_1)$ and $o_2 = g_2(i_2)$. Formula $l_{i_2} = l_{o_1}$ implies that the output variable o_1 should be unified with the input variable i_2, and hence the second component becomes $o_2 = g_2(o_1)$, suggesting the data flow between the two primitives.

The following constraint requires that the specification of each primitive component is satisfied.

$$\phi_{lib} = \bigwedge_{i=1}^{N} \phi_i(\overrightarrow{I_i}, O_i)$$

With the aforedefined constraints, the following θ defines the synthesis constraint. A valuation of L that satisfies θ defines a function equivalent to the original complex function f_c regarding the given input-output samples.

$$\phi_{func}(L, \overrightarrow{I}, O) = \exists Q, R, \phi_{wfp}(L) \wedge \phi_{lib}(Q, R) \wedge$$
$$\phi_{dataflow}(L, \overrightarrow{I}, O, Q, R) \wedge O = f_c(I)$$
$$\theta = \exists L \bigwedge_{\alpha_i, \beta_i \in E} \phi_{func}(L, \alpha_i, \beta_i)$$

There may be multiple models that can manifest the specified input-output behavior. Iterative refinement is used to remove bogus solutions. The idea is to use the SMT solver to generate new inputs (inputs that are not in E) such that there exist multiple f_s's producing different outputs from the same new inputs. The new inputs are then added to E and the synthesis process is repeated. The following constraint θ_{refine} is used to generate new inputs. A valuation of $\overrightarrow{I}$ is added to E. The whole process terminates when θ_{refine} cannot be satisfied.

$$\theta_{refine} = \exists L, L', \overrightarrow{I}, O, O', \theta(L) \wedge \theta(L') \wedge$$
$$\phi_{func}(L, \overrightarrow{I}, O) \wedge \phi_{func}(L', \overrightarrow{I}, O') \wedge O \neq O'$$

VI. Modeling Environment

In this section, we present the details of addressing various practical challenges in modeling environment. These are our contributions over [7].

A. Providing Expressive Primitive Components

System and library function implementations are often complex. Their models cannot be sufficiently constructed from simple arithmetic operations and statements. We need to provide more expressive primitive components.

1) Primitives for Data Structures.: System and library functions make heavy use of data structures. For the consideration of performance, data structure operations are often highly optimized, e.g. bit operations may be used to replace regular operations, software caches may be used to reduce response time. It is unnecessary to model the details of such optimized operations. We can effectively mask the unnecessary complexity if we abstract data structure operations to primitive components.

Recall the glibc file read example in Section III. Although the original implementation has over 700 lines of code [1] and it contains complex logic on file buffer control, caching, and memory management, it can be effectively modeled as array manipulations.

We observe that many system and library calls can be modeled using a few abstract data structures such as arrays, hash tables, and sets. For example, a pipe is essentially an array that is being accessed in a specific order. Abstracting the operations on these data structures to first-class primitives can substantially simplify the modeling process. SMT solvers do not explicitly support all these data structures. We hence *leverage the build-in support for the Array theory and project concrete data structures to the abstract Array type.* In particular, the SMT Array type is of the form (Array typeA typeB), where typeA is the type of the index and typeB is the type of the value. They can be customized to any types supported by the solver.

Array In C. We use Array to refer to arrays in SMT language and C_Array to refer to arrays in C language. To model array type in C, we instantiate typeA to Int. Therefore, the type descriptor of C_Array is (Array Int typeB). We show in Table II three modeled operations of C_Array: `subarray`, `concat`, and `indexof`. These operations are provided as primitive components for synthesis. Their constraints (specifications) contribute to the synthesis formula θ (Section V) through ϕ_{lib}. Operation `subarray` was used in the model for the glibc file read example.

Hash Table. Real world hash table and hash map implementations vary a lot, depending on the underlying hash functions, the implementation of hash buckets, and the logic for handling hash conflicts. We observe that at the behavioral level, all these implementations realize the same functionality of maintaining a one-to-one mapping of a key and a value, which can be directly described by the SMT Array theory. In particular, the type descriptor for hash tables is (Array typeA typeB), without instantiating typeA or typeB. The specification is omitted due to lack of space.

[1]The number includes those being called by glibc read to realize its functionality, down to the call depth of 3.

TABLE II
SPECIFICATION OF C_ARRAY.

Operation	Description	Constraint
`outarr = subarray(inarr, pos, size)`	Extracting a sub-array. The parameter `pos` denotes where to start extraction.	$(len(outarr) = min(len(inarr) - pos, size)) \wedge$ $(\forall k, 0 \leq k < len(outarr) \Rightarrow outarr[k] = inarr[pos + k])$
`outarr = concat(arrA, arrB)`	The `outarr` is the concatenation of `arrA` and `arrB`.	$len(outarr) = len(arrA) + len(arrB) \wedge$ $(\forall k, 0 \leq k < len(arrA) \Rightarrow outarr[k] = arrA[k]) \wedge$ $(\forall k, len(arrA) \leq k < len(outarr) \Rightarrow outarr[k] = arrB[k])$
`out = indexof(arr, i)`	The `out` is the *ith* element in `arr`.	$out = arr[i]$

TABLE III
SPECIFICATION OF SET.

Operation	Description	Constraint
`setB = insert(setA, element)`	Insert one element into the set.	$\forall k, k = element \Rightarrow setB[k] = true \wedge$ $k \neq element \Rightarrow setB[k] = setA[k]$
`setB = remove(setA, element)`	Remove one element from the set.	$\forall k, k = element \Rightarrow setB[k] = false \wedge$ $k \neq element \Rightarrow setB[k] = setA[k]$
`out = contains(setA, element)`	Check whether the set contains `element`.	$out = setA[element]$

Set and Linked List. Linked lists are another widely used type of data structure in system and library functions. Most linked list related behavior, e.g. inserting an element and looking for an element, can be modeled as set manipulation. We model a set as a mapping from a value to a boolean representing whether the value is an element of the set. Therefore, we instantiate typeB as Bool and the type descriptor for Set as (Array typeA Bool). The full specification of set operations is presented in Table III.

2) Higher Order Functions: We observe that many system and library functions can be modeled as high order functions that repeatedly apply a given function on the input. Examples include `atoi()` and `qsort()`. To synthesize these functions, we provide the following two high order function templates, and then reduce the problem to synthesizing the function that is provided to the high order function. We use h and f to denote the high order function and the provided function, respectively.

Higher Order Function Template (I): Transformation

```
h(f,arr) ={
    for(i = 0; i < len(arr); i++){
        outarr[i] = f(arr[i]);
    }
    return outarr;
}
```

Higher Order Function Template (II): Accumulation

```
h(f,arr) ={
    //setting acc[0], to be synthesized
    for(i = 0; i < len(arr); i++){
        acc[i+1] = f(acc[i], arr[i]);
    }
    return acc[len(arr)];
}
```

In the first template, f is a function that transforms individual elements in the input arr to the output $outarr$. In the second template, each output element is the accumulation of the input subsequence up to the current input element. To synthesize function f, we acquire input-output pairs of f as follows. In the case of transformation, each $arr[i]$ and $outarr[i]$ constitute an input-output pair of f. For accumulation, $(h(f, arr[0 : i-1]), arr[i])$ and $h(f, arr[0 : i])$ constitute an input-output pair of f. Given the original function f_c and i, $h(f, arr[0 : i - 1])$ can be obtained by executing f_c with input $arr[0 : i - 1]$. We have synthesized `atoi` using the accumulation template. The synthesized input function f is $f(x, y) = x * 10 + (y - 48)$ (48 is the decimal ASCII code of character '0'), with x the previously accumulated value and y the current input element.

3) Parameterized Primitives.: So far we have assumed that we know the precise functionality of each primitive such that the modeling process is merely to resolve the location variables in L. However, in many cases, we only know the form of the functionality but not the precise coefficients in the form. For example, when synthesizing the error code logic in file system calls, we know constant assignments need to be provided as the primitives (because error codes must be set through such assignments), but we do not know what are those constants. In some cases, we know a variable ought to be updated, but we do not know if the update primitive is a plus one or a minus one operation. Similar examples include comparisons with constants.

We address the issue by parameterizing primitive components. Each primitive component has a set of parameters, depending on which the precise functionality of the component varies. Note that a parameter of a primitive g_i is different from an input of g_i. The parameters of g_i determine what function g_i actually is. The inputs of g_i determine the corresponding outputs with the specific functionality. For example, an addition-with-constant primitive denoted as $o = \text{add}^c(i)$ means $o = i + c$ with i being the input and c being the parameter. More examples of parameterized primitives can be found in Table I, with superscripts representing the parameters.

Let P be the set of all parameters for the primitives. The modeling process now is to resolve both L and P. The essence of introducing P is to leverage the SMT solver to search for the precise form of a primitive. An alternative is to explicitly provide all the possible instantiations of a parameterized primitive. However, such an approach requires prior knowledge of all instantiations and it results in a very large set of primitives.

B. Reducing Search Space with Type Constraints

With the non-trivial set of primitive components and the introduction of parameterization, the search space is large. We develop an optimization that helps reduce the search space. Specifically, we leverage type checking by explicitly encoding type constraints. For each variable x in $P \cup Q \cup R \cup \overrightarrow{I} \cup \{O\}$, we use $type(x)$ to represent the type of x. A valid program should have consistent types for each variable in the program. We have the following type constraint.

$$\psi_{type} = \bigwedge_{x,y \in P \cup Q \cup R \cup \overrightarrow{I} \cup \{O\}} type(x) \neq type(y) \Rightarrow l_x \neq l_y$$

Intuitively, it dictates variables with different types cannot be unified. A more complex type lattice could be adopted, but we have not seen the necessity. The type constraint now becomes part of the synthesis constraint.

C. Handling Loops

Loops are heavily used in system and library functions. However, it is known to be difficult to directly synthesize loops [17] due to the fact that the underlying SMT solver cannot reason about loops. We handle loops in two ways. The first one is similar to handling higher order functions. We provide pre-defined loop templates and reduce the problem to synthesizing the loop body.

In many cases, we are able to avoid synthesizing loops by abstracting the iterative semantics to data structure primitives. One example is the glibc file read function. Even though its implementation contains loops, the `subarray()` primitive abstracts away such loops. Similarly, with our hash table primitives, we abstract away all the loops related to hash value computation and hash bucket traversal, etc.

D. Selecting Primitive Components

The basic synthesis procedure introduced in Section V requires a set of primitive components to begin with, which are provided by the user based on their domain knowledge. The set determines the search space of the synthesis procedure. If the provided primitive components are insufficient, synthesis will fail. On the other hand, providing too many primitive components reduces efficiency.

We propose Algorithm 1 to select primitive components. Initially we only provide the very basic components: `add`, `const`, `greater-than`, `if-else`. These components are most commonly used in environment functions. If the synthesis procedure fails with the provided component set P, one additional primitive component is randomly selected and added. This process continues until the synthesis procedure succeeds or time runs out. Note that providing excessive primitives does not affect the capability of synthesizing correct models. Unused primitives become dead code. One example is line 9 in Fig. 3. We currently do not eliminate dead code.

We also use the signature of the synthesized function to determine whether data structure primitives are considered when additional primitive components are required (see Algorithm 1).

Algorithm 1 may not always succeed. If that happens, users have to interfere the selection of primitive components. More discussion will be presented in Section VIII. Note that in Algorithm 1, P and A are multisets, which could contain multiple instances of the same primitive component.

Algorithm 1 Modeling Environment

```
1: let multiset $P$ be {add, const, greater-than, ifelse}
2: let multiset $A$ be {add, minus, const, greater-than, less-than, equal, ifelse, ... }
3: let $Sig$ be the signature of the synthesized function
4: if $Sig$ uses T * where T is a primitive type (e.g. char, int) then
5:     put array primitive components into $A$
6: end if
7: if $Sig$ uses U * where U is a user-defined type then
8:     put set and hashtable primitive components into $A$
9: end if
10: while not successful and not timeout do
11:     try to synthesize the program with $P$
12:     if successful then
13:         break
14:     end if
15:     Randomly choose one component from $A$ and put into $P$
16: end while
```

E. An Example of Iterative Modeling

Next we use an example to illustrate the iterative modeling process. The original function f_c is shown below.

```
int f(int x){
    if(x<10)
      return -1;
    else
      return 0;
}
```

The provided primitives include parameterized constant assignments and conditional statements. The initial input-output specification E contains only one pair $\{(11, 0)\}$. Table IV presents the modeling process. The first column shows the iteration number. The (L, P) and (L', P') columns show the solution to the iterative refinement constraint θ_{refine} in Section V. Note that we add in P and P' to denote the parameters that need to be synthesized. Intuitively, in each iteration, two programs are constructed, both satisfying the input-output specification, but having different behavior on some input. The input that distinguishes the two constructed programs is added to E and another iteration starts. For each program, the `ordering` column represents the raw solution from the solver, and the `program` column shows the corresponding C code. Observe that after two iterations, the process terminates at the final solution (that is, (L', P') cannot be resolved). Even though the final program is syntactically different from the original one, it is semantically equivalent.

VII. Validating Models

After acquiring the synthesized function f_s, we need to check whether f_s indeed conforms with the real-world im-

TABLE IV
THE ITERATIVE SYNTHESIS OF A SAMPLE PROGRAM. PRIMITIVE $o = \text{CONST}^c$ DENOTES A PARAMETERIZED CONSTANT ASSIGNMENT THAT ASSIGNS c TO o.

Iter.	E	(L,P)		(L',P')	
		ordering	program	ordering	program
1	(11,0)	o0 = input o1 = const^{9} o2 = const$^{(-1)}$ o3 = lt(o0,o1) o4 = const^{0} o5 = ite(o3,o2,o4) return o5	int f(int x){ if (x< 9) return -1; else return 0; }	o0=input o1=const^{9} o2=lt(o1,o0) o3=const^{0} o4=constc$^{(-1)}$ o5=ite(o2,o3,o4) return o5	int f(int x){ if (9 < x) return 0; else return -1; }
2	(11,0) (9, -1)	o0=input o1=const^{10} o2=lt(o0,o1) o3=const$^{(-1)}$ o4=const^{0} o5=ite(o2,o3,o4) return o5	int f(int x){ if (x<10) return -1; else return 0; }	o0=input o1=const^{0} o2=const$^{(-1)}$ o3=const^{10} o4=lt(o3,o0) o5=ite(o4,o1,o2) return o5	int f(int x){ if (10<x) return 0; else return -1; }
3	(11,0) (9, -1) (10,0)	o0=input o1=const^{9} o2=lt(o1,o0) o3=const^{0} o4=constc$^{(-1)}$ o5=ite(o2,o3,o4) return o5	int f(int x){ if (9 < x) return 0; else return -1; }		

plementation f_c. Due to the complexity of f_c, it is often impractical to perform any manual or automated theorem proving. In this paper, we resort to testing. We perform black-box testing by generating inputs from the description of f_c. Such description is available from the documentation of f_c. We apply equivalence class partitioning [18] and pair-wise testing [19] during test generation. The generated inputs are executed in both f_s and f_c. If there is any input t that produces different outputs in the two functions, we enhance E with $(t, f_c(t))$ and perform another iteration of modeling. Note that validation is different from refinement, which tries to converge on a single program that conforms to the given I/O pairs. However, the converged version could be incorrect. The validation step hence validates results after convergence.

Potentially, we would be able to perform more thorough testing by white-box approaches. We have tried dynamic test generation for f_c. Unfortunately, the engines we have tried [3], [4] failed to perform symbolic execution inside system calls. We leave it as our future work to further investigate this issue.

VIII. EVALUATION

We have implemented a prototype using Perl. Z3 [20] is the underlying SMT solver, which supports integer and bitvector arithmetics as well as array. The SMT2 language that Z3 supports also extends first-order logic with if-then-else function.

A. Synthesis results

The first set of functions we synthesized are from Linux system calls. In particular, we focus on system calls that are used in coreutils-6.11 [2]. They are largely file system related, including `read`, `write`, `seek`, `chmod`, `chown`, `stat` and a few variations of these system calls[3].

TABLE V
SYNTHESIS RESULTS.

Function	#Prim	Time	#IO pairs init/final	LOC in C		Manual interference
				f_s	f_c	
read	8	10s	2/13	15	782*	subarray
write	11	1s	2/4	21	650*	replacearray
stat	4	1s	2/3	4	588*	n/a
seek	8	8s	2/8	11	200*	equal
chmod	5	1s	2/7	8	571*	n/a
chown	6	2s	2/7	9	1059*	n/a
atoi	8	77m	2/4	11	307*	loop template multiply
malloc	5	1.5m	3/6	10	431*	rem
max	6	3s	2/8	9	$4^{\S}$	n/a
min	6	2s	2/5	9	$4^{\S}$	n/a
sign	8	8m	2/5	13	$4^{\S}$	n/a
absolute	5	2s	2/5	8	$3^{\S}$	n/a

*Since they call many other functions to realize their functionality, we include the lines of code of those functions down to call depth of 3. §These functions are implemented in bitwise operations.

We further look into the most commonly used glibc [21] library functions in coreutil. A large portion of these functions are system call wrappers. They can be synthesized in the same way as synthesizing raw system call functions and thus not presented. Apart from these wrapper functions, we also synthesize `atoi` and the real allocated size of `malloc`.

Lastly, we experiment on some highly optimized functions. In particular, we choose functions from [22]. These functions are implemented using bitwise operations. While being very efficient, these functions cause difficulty for both human understanding and program analysis. Using these highly optimized functions as reference, we synthesize equivalent functions that are easier to understand and analyze. In this category, we have synthesized `max`, `min`, `sign` and `absolute`.

The results of our synthesis experiment are shown in Table V. The second column shows the number of primitives in the synthesized f_s. The third column shows the time and the fourth shows the size of input-output pairs at the beginning and after the iterative refinement process terminates. Note that the difference between the two sizes is the number of refinement iterations. The next two columns show the lines of code of the generated C programs and the original C implementations. The last column shows manual interference. When specified, we need to manually select these components for successful synthesis.

Among the 12 functions in Table V, 7 functions are automatically synthesized using Algorithm 1 without any manual interference. For the other 5 functions, we have to manually provide primitive components for the synthesis procedure to succeed. For example, we need to provide `multiply` to synthesize `atoi` function. We prefer not to include these primitives into the default set because using them in synthesis is expensive compared to others.

Observe that even though many of the original implementa-

[2]http://www.gnu.org/s/coreutils/
[3]We do not present results for such variations due to similarity.

tions are complex (i.e. mostly a few hundred lines), we are able to acquire very simple models (i.e. mostly less than 20 lines). We perform both testing and manual inspection to validate the correctness of the generated models. Also observe that the technique does not require a large initial input-output set. It is able to generate new inputs through refinement and quickly converges. The initial input-output set is randomly generated.

The current timeout for the synthesis algorithm is 10 minutes. If the algorithm fails to generate a model in 10 minutes, we manually add some more primitives according to our understanding of the functions. We never needed to add more than two primitives. For those that needed manual interference, the third column of Table V shows the synthesis time after we manually added the primitives. For `atoi()`, we have to set the timeout to 2 hours after adding the `multiply` primitive as reasoning about multiplication constraints is very expensive for the SMT solver. Furthermore, the constant used in multiplication is not known a priori and is synthesized.

B. Improving symbolic execution with synthesized models

Symbolic execution has been successfully used to explore program paths and automatically generate test inputs. Due to the complexity of system call code, none of the existing engines can symbolically execute system calls together with the application program. When a system call is encountered in symbolic execution, some approximation has to be introduced. The approximation can be avoided using our synthesized system call models.

We use KLEE [4] as the symbolic execution engine for our experiment. The authors of KLEE have manually written some models for system calls. We do two sets of comparison. First, we examine whether our synthesized system call models could help symbolic execution in KLEE. We compare the instruction coverage results between KLEE with no system call models and KLEE with our synthesized models. Second, we examine the quality of our synthesized models by comparing them with KLEE's manually written models.

We use coreutils-6.11 as our subject programs. As KLEE could continuously run for a long time, trying to explore all program paths, we set a time-out at 10 minutes for each program. The comparison results between no-model and the synthesized models are shown in Table VI. As shown in Table VI, with the help of our synthesized models, KLEE gets better instruction coverage for most of these programs. The maximum improvement is 43%. The average improvement over the entire set of programs used (71) is 2.7%. There are a few cases where the instruction coverage is worse with the synthesized models. The reason is that given the same time bound, symbolic execution could focus more on the application program when there are no system call models, leading to better instruction coverage.

We have also compared our synthesized models with KLEE's original models. Out of the 71 programs we have run, our models get better results in 23 programs. KLEE's models perform better in 27 programs. For the rest 21 programs, our models perform exactly the same as KLEE's models.

TABLE VI
INSTRUCTION COVERAGE RESULT ON COREUTILS-6.11 (KLEE WITH OUR SYNTHESIZED MODEL VS. KLEE WITH NO SYSTEM CALL MODEL)

Instruction coverage difference (synthesized model - no model)	#Programs
$> 30\%$	1
$20\% \sim 30\%$	4
$10\% \sim 20\%$	6
$2\% \sim 10\%$	18
$-2\% \sim 2\%$	27
$-10\% \sim -2\%$	10
$-20\% \sim -10\%$	3
$-30\% \sim -20\%$	2
$< -30\%$	0

The average difference is negligible. That is to say, our models are as good as the manual models. We argue that potentially our analytical system can be deployed on different environments and generate the models automatically, reducing human efforts.

C. Enhancing failure surviving with our synthesized models

We have modeled the real allocated size of glibc `malloc` and evaluated the effectiveness of Rx with and without the model. Without the model, Rx adds a fixed-size padding (4 bytes by default and the size can be configured by users). With the model, Rx dynamically queries it to determine the proper padding size. The core of the model generated is as follows with `x` the requested size.

```
if(x<12) return 16;
else return (x+11) & 0xfffffff8;
```

It means if the size is smaller than 12, `malloc` returns 16 bytes. Otherwise, it increases the size by 11 bytes and then masks off the least 3 bits.

We evaluate Rx with a buffer overflow bug in `bc`, a numeric processing program with more than 8,000 lines of code. The overflowed array stores four pointers with four bytes each. When bc requests 16 bytes for the array, the glibc library allocates 24 bytes, including the 8 bytes padding according to the model. The program crashes once the array is overflowed for more than 8 bytes. With the default 4-byte padding, Rx fails to recover from the failure since glibc still returns 24 bytes due to the implicit byte alignment. By consulting the model, Rx selects a 8-byte padding, which makes difference on the real size of allocated memory, and successfully survives the failure.

IX. RELATED WORK

Our basic synthesis constraints encoding is based on the oracle-based program synthesis work in [17], [7]. However, many challenges arise when applying the basic approach in [17] to synthesize real world library/system calls. In this paper, we have focused on solving these challenges.

Apart from [17], [7], many other work have applied program synthesis to different application domains. In [23], the authors synthesize string processing functions in spreadsheets. In [24], the authors construct program inversions using inductive program synthesis. In [25], Gulwani et. al. have also applied program synthesis to geometry constructions.

There have been few works on automatic environment modeling. Instead, manually crafted models for system calls and common library calls are used in some symbolic execution engines [4], [3], [26]. On the other hand, DART [27] and CUTE [28] perform unit testing and over-approximate the effects generated by environment. An input from environment may take any value within its domain. In model checking, Tkachuk and Dwyer [29] generate over-approximated environment models by analyzing the environment source code. Under-approximation by simply executing the environment code has also been used in previous research on symbolic execution [30].

In unit testing, mock objects [31], [32] are widely used to simulate the execution environment of the unit being tested. There also exist some mocking frameworks [33], [34] that can partially automate the task of creating mock objects. However, manual effort is still required to create mock objects that implement more than dummy interfaces. The manual effort in creating mock objects can be mitigated using our synthesis technique.

In [35], function summaries are generated and reused in symbolic execution. However, this approach cannot be applied to system call code due to the high complexity of system calls. Moreover, symbolic execution, as a whitebox approach, can hardly abstract away any optimization code in the implementation. The work in [36] and [37] infer behavior models of functions in the form of constraints. These constraints serve as loose specifications of the modeled functions but do not express exact input-output relationship.

X. Conclusion

We propose a technique that shows the feasibility of modeling software environment, including system and library functions used by a program. Given the binary implementations of these functions, it first acquires input-output specification by sampling their executions. An SMT solver is used to construct the model, a simple C program, that satisfies the specification. The model is iteratively refined though automatically generated counter-examples. The technique addresses problems specific to modeling environment, such as modeling data structures, reducing search space, and handling loops. The experiments show that a number of commonly used file-system-related functions can be precisely modeled. Library functions, including `atoi()` and `malloc()`, are also modeled. It is also shown that the generated models can be used to improve symbolic execution and failure tolerance.

Acknowledgement

This work was partially supported by National Science Foundation (NSF) grants CCF-0953759 (CAREER Award), CCF-0845870, CCF-1218358, CCF-1218993, and a Ministry of Education research grant MOE2010-T2-2-073 (R-252-000-456-112 and R-252-100-456-112) from Singapore.

References

[1] H. Agrawal and J. Horgan, "Dynamic program slicing," in *PLDI*, 1990.

[2] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *NDSS*, 2005.

[3] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A new approach to computer security via binary analysis," in *ICISS*, 2008.

[4] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008.

[5] J. Wei, "Improving operating systems security: Two case studies," *PhD Dissertation, Georgia Institute of Technology*, 2009.

[6] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: treating bugs as allergies—a safe method to survive software failures," in *SOSP*, 2005.

[7] S. Jha, S. Gulwani, S. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *ICSE*, 2010.

[8] M. Weiser, "Program slicing," in *ICSE*, 1981.

[9] A. Myers, "JFLow: Practical mostly-static information flow control," in *POPL*, 1999.

[10] J. Gray, "Why do computers stop and what can be done about it?" in *SRDS*, 1986.

[11] Q. Gao, W. Zhang, Y. Tang, and F. Qin, "First-aid: surviving and preventing memory management bugs during production runs," in *EuroSys*, 2009.

[12] "Squid," http://www.squid-cache.org/, 2011.

[13] "HTTP Specification," http://www.w3.org/Protocols/Specs.html, 2011.

[14] "The GNU C Library," http://www.delorie.com/gnu/docs/glibc/libc_31.html, 2011.

[15] V. Lvin, G. Novark, E. Berger, and B. Zorn, "Archipelago: trading address space for reliability and security," in *ASPLOS*, 2008.

[16] C. Lattner and V. Adve, "Automatic pool allocation: improving performance by controlling data structure layout in the heap," in *PLDI*, 2005.

[17] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, "Synthesis of loop-free programs," in *PLDI*, 2011.

[18] E. Weyuker and B. Jeng, "Analyzing partition testing strategies," *IEEE Transactions on Software Engineering*, vol. 17, no. 7, 1991.

[19] Y. Lei and K. Tai, "In-parameter-order: a test generation strategy for pairwise testing," in *HASE*, 1998.

[20] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *TACAS*, 2008.

[21] "GNU C library," http://www.gnu.org/s/libc/, 2011.

[22] H. Warren, *Hacker's delight*. Addison-Wesley Professional, 2003.

[23] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *POPL*, 2011.

[24] S. Srivastava, S. Gulwani, S. Chaudhuri, and J. S. Foster, "Path-based inductive synthesis for program inversion," in *PLDI*, 2011.

[25] S. Gulwani, V. Korthikanti, and A. Tiwari, "Synthesizing geometry constructions," in *PLDI*, 2011.

[26] K. Havelund and T. Pressburger, "Model checking java programs using java pathfinder," *STTT*, vol. 2, no. 4, 2000.

[27] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *PLDI*, 2005.

[28] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *ESEC/FSE*, 2005.

[29] O. Tkachuk and M. B. Dwyer, "Adapting side effects analysis for modular program model checking," in *ESEC/FSE*, 2003.

[30] C. S. Păsăreanu, N. Rungta, and W. Visser, "Symbolic execution with mixed concrete-symbolic solving," in *ISSTA*, 2011.

[31] T. Mackinnon, S. Freeman, and P. Craig, "Endo-testing: unit testing with mock objects," *Extreme programming examined*, 2001.

[32] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes, "Mock roles, not objects," in *OOPLSA*, 2004.

[33] "Google C++ Mocking Framework," http://code.google.com/p/googlemock/, 2011.

[34] "Easymock," http://easymock.org/, 2011.

[35] P. Godefroid, "Compositional dynamic test generation," in *POPL*, 2007.

[36] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1-3, 2007.

[37] L. Mariani and M. Pezzè, "Dynamic detection of cots component incompatibility," *IEEE Software*, vol. 24, no. 5, 2007.