Effective Generation of Interface Robustness Properties for Static Analysis

Mithun Acharya, Tanu Sharma, Jun Xu, Tao Xie
Department of Computer Science
North Carolina State University
Raleigh NC USA 27695
{mpachary, tsharma}@ncsu.edu, {junxu, xie}@csc.ncsu.edu

Abstract

A software system interacts with its environment through system interfaces. Robustness of software systems are governed by various temporal properties related to these interfaces, whose violation leads to system crashes and security compromises. These properties can be formally specified for system interfaces and statically verified against a software system. But manually specifying a large number of interface properties for static verification is often inaccurate or incomplete, apart from being cumbersome. In this paper, we propose a novel framework that effectively generates interface properties for static checking from a few generic, high level robustness rules that capture interface behavior. We implement our framework for an existing static analyzer with simple data flow extensions and apply it on POSIX-API system interfaces used in 10 Redhat-9.0 open source packages. The results show that the framework can effectively generate a large number of useful interface properties from a few generically specified rules.

1 Introduction

Robustness of software systems are governed by various temporal rules related to interfaces, such as rules for proper exception handling. Traditional robustness testing [4] has been used to detect robustness problems by generating invalid system inputs. However, the implicit exceptional returns through system interfaces cannot be easily generated. Proper exception handling and other robustness rules can be specified as formal properties and statically verified [1, 2] against a software system. Specifying formal properties require the knowledge of source code and interface specifications such as return values and exceptions. Thus manually specifying a large number of interface properties for static verification is often inaccurate or incomplete, apart from being cumbersome. To address these issues, we propose a novel framework that generates interface properties from a few generic, high level robustness rules that capture interface behavior. Generic robustness rules are specified at an abstract level that needs no knowledge of the source code, system, or interfaces. These generic rules are then translated by our framework into concrete properties, verifiable by static analyzers. The translation uses interface and source code level information that are specified by interface implementers with low one-time effort. We implement our framework for an existing static analyzer with our data flow extensions and apply it to the well known POSIX-API system interfaces. The generated concrete properties are used to statically check 10 Redhat-9.0 packages for robustness violations. This paper makes the following contributions:

- We propose a framework for effectively generating interface properties from a few generic, high level robustness rules.
- We implement the framework and apply it to the the well known POSIX-API system interfaces. Roughly, 1000 concrete formal properties ($>$ 30,000 lines) were generated from 6 generically specified rules ($<$ 60 lines) for 280 POSIX-API interfaces, highlighting the effectiveness of our approach. We statically check 10 Redhat-9.0 packages against the generated interface robustness properties by using an existing static analyzer with our data flow extensions.

The rest of this paper is organized as follows. Section 2 presents our framework for generating interface robustness properties. Section 3 presents our experimental results and Section 4 concludes the paper with future work.

2 Robustness Property Generation

The goal of our framework is to allow developers to specify robustness rules generically without the knowledge of the system, language, and interfaces. To abstract away these details from developers, we make use of two key observations about interfaces and their robustness rules. The first observation is that related interfaces have similar structural *elements* (such as function parameters, return values on success/failure, and error flags) when specified at a certain abstract level. The second observation is that most interface robustness violations are temporal orderings of certain *actions* (such as invoking an interface, checking interface returns for success/failure, dereferencing an interface return, and passing an interface return to a function) that could be performed on an interface or its elements.

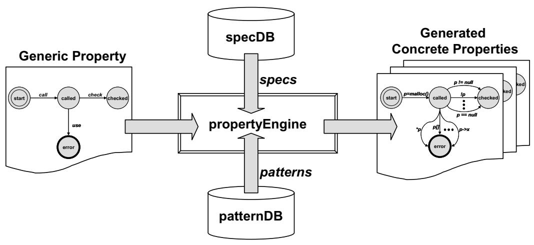


Figure 1. A Framework for Concrete Property Generation

The overview of our framework is shown in Figure 1. Developers define generic rules at a high level over interface elements and actions, without the details of interfaces and source code. The details of interfaces are stored in a specification database (`specDB`) and the source-code details of interface elements and actions are stored in a pattern database (`patternDB`). The generic rules are translated into concrete properties by a `propertyEngine` that queries `specDB` for interface-level information and `patternDB` for source-code level, programming-language specific information. In the subsections that follow, we identify the elements and actions that characterize an interface and show how generic rules are defined and concrete properties are derived from them, with a simple illustrative example.

2.1 Interface Characterization

A set of interfaces implemented for a specific purpose have similar structural details at a high level. We characterize an interface with its structural elements and actions that can be performed on them. The characterization allows us to systematically store the interface and language patterns for these interfaces in a database. For a given interface, the `propertyEngine` can query the database on the keywords of elements or actions to get low-level details.

For any interface $i \in \mathcal{I}$, where $\mathcal{I}$ is a related family of interfaces, we define an interface specification as

$$spec(i) = \{is(i), rs(i), ss(i), \mathcal{R}, \mathcal{S}, \mathcal{Z}\}$$

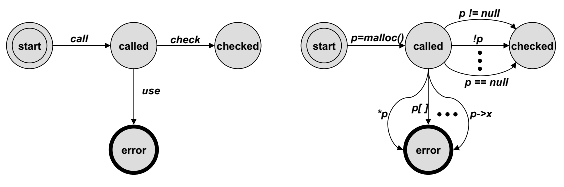
$is(i)$ is the set of input parameters passed to the invocation of i, $rs(i)$ is the result set, the set of variables that store the return values of interface execution and $ss(i)$ is the status set, the set of variables that store the failure status or type of failures of the interface. Any variable $v \in is(i) \bigcup rs(i) \bigcup ss(i)$ is called the *element* of i. $\mathcal{R}$ is a mapping from $rs(i)$ to $\mathcal{Z}$, while $\mathcal{S}$ is a mapping from $ss(i)$ to $\mathcal{Z}$, where $\mathcal{Z}$ holds the values that members of $rs(i)$ and $ss(i)$ would assume on success or failure of interface execution. For a related family of interfaces, $\mathcal{I}$, we define an $action$ set as a set of actions that can be performed on the interface itself or its elements.

For example, $\mathcal{I}$ could be POSIX-API interfaces and $i \in \mathcal{I}$ could be `malloc`. Before a statement such as `p = malloc(x)` is executed in a program, $is(\texttt{malloc})$ is $\{x\}$, $rs(\texttt{malloc})$ is $\emptyset$ and $ss(\texttt{malloc})$ is $\emptyset$. After the statement execution, $rs(\texttt{malloc})$ is $\{\texttt{p}\}$ and $ss(\texttt{malloc})$ is $\{\texttt{p}\}$. For `malloc`, the return and the failure/success indicator are the same. If the `malloc` call succeeds, `p` is a memory pointer (say, `mptr`) and then $(\texttt{p}, \texttt{mptr}) \in \mathcal{R}$. If it fails, `p` assumes value `NULL` and $(\texttt{p}, \texttt{NULL}) \in \mathcal{R}$. Because the result set and status set are the same for `malloc`, we have $\mathcal{S} = \mathcal{R}$. The set $\mathcal{Z} = \{\texttt{mptr}, \texttt{NULL}\}$ holds the success/failure indicators for the `malloc` API.

The $action$ set for the POSIX-API interfaces comprises $alias$, $call$, $check$, $FALSE$, $failure$, $free$, $pass$, $return$, $success$, $TRUE$ and use. The meanings of the actions are self-evident. Whenever a return variable is aliased, the action performed on $v \in rs(i)$ (the return value of the interface execution) is $alias$. The action of invoking the interface is $call$. The action of checking the interface return value or status against members in $\mathcal{Z}$ is $check$. If $check$ fails, the action is $FALSE$ (for example, if the check `if (p==NULL)` fails, the action is $FALSE$ and `p` assumes non `NULL` value). If $check$ succeeds, the action is $TRUE$. The interface execution can either be a $failure$ or $success$. When the interface return variable is passed to another function, a $pass$ action is said to be performed. When the function in which the interface is executed returns, the action performed is $return$. Finally, when the return value is used in program expressions, the action is use. While interface specificaiton details are stored in the `specDB`, source code details for each of the actions and elements are stored in the `patternDB`. We next present a simple example illustrating how concrete robustness properties can be formed by defining generic rules over interface elements and actions.

2.2 Example

Generic rules for an interface $i \in \mathcal{I}$ are defined over the members of the $action$ set of $\mathcal{I}$. A generic rule is some ordering constraints on the members of the $action$ set. We use a Finite State Machine (FSM) to graphically represent a generic rule. The FSM has a start state and an error state as well as other user-defined states. A sequence of actions that violates the robustness property represented by the FSM takes the FSM to the error state. The edges of the FSM are members from the $action$ set. For example, for the `malloc` interface, the use action should always be preceded by the $check$ action. The FSM for such a rule is shown in Figure 2(a). Generic robustness rules are currently manually specified.



(a) ***check*** **should always precede** ***use*** **generic property**

(b) Concrete ***UseBeforeCheck*** **property for the** ***malloc*** **API call**

Figure 2. Generic *UseBeforeCheck* property and the corresponding concrete property for `malloc` API

To generate concrete property for `malloc`, `propertyEngine` queries the `specDB` for POSIX-APIs to obtain details about `malloc` and learns that the return type of `malloc` is a pointer on success and `NULL` on failure. Based on this information, the `propertyEngine` constructs a query to the `patternDB` that comprises the keyword $check$, the data type of the return variable (being a pointer in this case), and values on success and failure. The `patternDB` processes this query and returns patterns for all the possible ways a pointer variable can be checked against `NULL` (or not `NULL`) (`if(p==NULL)`, `if(p)`, `if(!p)`, etc.). The `propertyEngine` expands the generic keyword $check$ to language and interface specific patterns. The same procedure applies to the keyword $call$ (`if(p=malloc(...))!=NULL`, `p=malloc(...)`, etc.) and use (`p->x`, `*p`, `p[x]`, etc.). The generated concrete property is shown in Figure 2(b).

API	parameter list	return type	return value		errno
			on success	on failure	
chmod	const char * path , …	int	0	-1	EPERM, …
open	const char * pathname, …	int	fd	-1	EEXIST, …
malloc	size_t size	void *	pointer	null pointer	
fsetpos	FILE * stream , …	int	0	-1	EBADF, …
remove	const char * pathname	int	0	-1	EFAULT, …

Figure 3. Selected Entries from the `specDB` for POSIX-APIs (simplified for presentation)

3 Evaluation

We have manually built the `pattern-DB` for the `C` language (using Abstract Syntax Tree notation to represent language patterns) and `specDB` for 280 POSIX-API interfaces (see Figure 3). The `pattern-DB` is a constant file specific to each programming language and contains the source code information for different language operations (e.g., dereference, check) that can be performed on simple and complex or derived data types. The `pattern-DB` can also be built for languages other than C.

We have built the framework to convert high-level, language-independent robustness and safety rules into concrete properties that can be directly used by a static checker. We have also applied the initial prototype to 10 Redhat 9.0 open source packages that use POSIX-API interfaces. We specified six simple generic properties, all of which pertain to the safe usage of memory pointers that hold the interface return values. From these generic rules, the `propertyEngine` generated more than a thousand useful concrete robustness properties ready for static verification. Figure 4 shows a highly simplified concrete property generated for the `opendir` interface. The concrete property presented governs the correct usage of pointer return variables. Simplified details (shown in the box) are shown only for one keyword, $check$, which is split into multiple edges and states by the `propertyEngine` using return value information from the specification database.

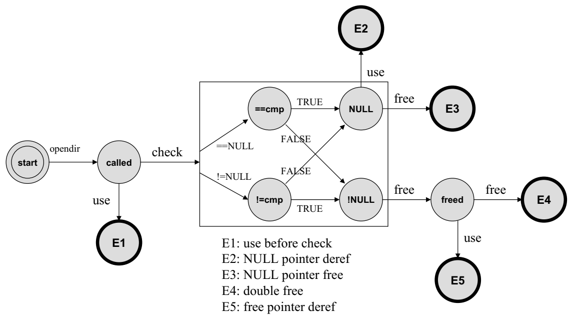


Figure 4. Simplified concrete property for `opendir` interface

We used our framework to analyze open source packages written in `C` mostly from the Redhat-9.0 distribution. In our experiments, we used a Pentium IV machine with 2.8GHz processor speed and 1GB RAM running on the Fedora Core 3 2.6.9-1.667smp kernel. In the experiments, we selected 10 widely used open source packages from the Redhat-9.0 distribution; these 10 packages include near 100K lines of `C` code. For static verification, we used a publicly available static analyzer called MOPS [2], which employs pushdown model checking to detect control flow errors at compile time. It constructs a Push Down Automaton (PDA) for a `C` program from its Control Flow Graph (CFG). It then generates a new PDA by composing the property FSM to be checked and the program PDA. The new PDA is model checked [3] to see if there is any path in the program that takes the new PDA to an error configuration. If there exists such a path in the program, the static checker reports the path as the error trace that violates the concrete robustness property.

The generic properties we specified are data-flow sensitive, i.e., they are dependent on the value of the return variable along different execution paths. Because the basic MOPS static checker is data-flow insensitive, it assumes that a given variable might take any value. Therefore, it assumes that both branches of a conditional statement may be taken and that a loop may execute anywhere between zero to infinite iterations. Because exception handling procedures are usually characterized by conditional constructs that check the return value of an API call, we write extensions to the static analysis procedure in order to make it possible to track the value of variables that take the return status of an API call along different branches of conditional constructs. For each possible execution sequence, our extensions associate a value to the variable that is being tracked using pattern matching. The concrete properties (in the form of FSMs) generated by the `propertyEngine` are given to the static analyzer enhanced with our data flow extensions. We evaluate the effectiveness and usefulness of our framework as follows.

Effectiveness: A user only needs to specify a small set of generic properties at a high level. The `propertyEngine` automatically generated more than a thousand formal concrete properties ($> 30{,}000$ lines) from 6 generically specified rules (< 60 lines) for 280 POSIX APIs. For static verification, we selected 60 critical API calls that are mainly used for memory management, file and string I/O, permission management, setting privileges, and spawning processes. These APIs are frequently used in applications and their safe and robust usages are critical for reliability and security. We then generated concrete properties for them across 6 generic properties using our property generation framework. For these 60 APIs, more than 300 concrete rules were generated and they were checked against the 10 Redhat-9.0 open source packages for robustness violations.

Usefulness: Table 1(a) presents the total number of robustness property violations our tool found for each of the checked packages. We found around 200 robustness violations in 10 Redhat-9.0 open source packages. We have shown the API-level violation breakdown for one selected package (`SysVinit-2.84-13`) in Table 1(b). Of the 60 analyzed APIs, 19 of them gave violations with this package.

Table 1. Robustness violations detected for the open source packages

package	# errors
ftp-0.17-17	18
ncompress-4.2.4-33	6
routed-0.17-14	15
rsh-0.17-14	9
sysklogd-1.3.31-3	27
sysstat-4.0.7-3	24
SysVinit-2.84-13	64
tftp-0.32-4	14
traceroute-1.4a12-9	7
zlib-1.1.3-3	4

(a) Overall Errors 10 Packages

API	# errors	API	# errors
fdopen	1	chdir	2
closedir	1	fstat	3
fflush	2	malloc	1
fileno	1	open	2
fputc	1	fclose	12
fputs	2	putchar	1
fseek	2	unlink	4
ftell	1	write	4
getpwuid	1	setuid	1
close	26		

(b) Errors from SysVinit-2.84-13

4 Conclusions and Future Work

We have proposed a framework for effectively generating a large number of interface properties from generic, high level robustness rules. We have implemented our framework and applied it to the well known POSIX-API system interfaces. We statically checked 10 Redhat-9.0 packages against the generated interface robustness properties by using an existing static analyzer with our simple data flow extensions. Although we have applied our approach only to POSIX-APIs, it can be easily applied to any set of interfaces. The `specDB` could be instantiated for other families of interfaces. Likewise, `patternDB` can be instantiated for languages other than C. Interface specifications (required for the generation of concrete properties) and many system-specific properties that govern the robustness, security and performance of software systems are often not documented by developers. We are currently working on a framework to automatically infer system-specific interface specifications and implicit temporal properties from program source code. We use a model checker to generate static traces related to the interfaces. The interface specifications and temporal properties are inferred from these traces using statistical analysis and certain heuristics.

References

[1] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *Proc. of IEEE Symposium on Security and Privacy*, pages 143–159, 2002.

[2] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, pages 235–244, 2002.

[3] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking push down systems. In *Proc. of Computer Aided Verification (CAV)*, pages 232–247, 2000.

[4] P. Koopman and J. DeVale. The exception handling effectiveness of POSIX operating systems. *IEEE Transactions on Software Engineering*, 26(9):837–848, 2000.