Computer Aided Extrinsic Robustness Verification

Christèle Faure, christele.faure@safe-river.com

***Abstract*— This paper answers an industrial question: "Given the specification of input values, is it possible to verify that the source code of a program is robust with respect to erroneous inputs and memory alterations?".**
We show that such verification is possible but quite complex to perform manually and we propose a semi-automatic solution. Our work is original in two ways: a new notion of software robustness is defined, enforced and verified, and we make use of a static tool in a non standard manner.

I. Introduction

We consider the following problem: given the specification of input values, verify that the source code of the program is robust with respect to spurious inputs and memory alterations? This is not a standard (named intrinsic) robustness problem because it does not deal with errors caused by the software itself. The underlying property is named extrinsic robustness because it handles errors caused by interactions between the software and its environment (hardware and software). The existing way to address this problem is based on manual code reviews. We have developed a semi-automatic approach based on static analysis [3] to solve it formally. From a definition of the extrinsic robustness in II, we derive enforcement mechanisms in III, experiment our semi-automatic verification method on industrial software in IV and conclude in V.

II. Extrinsic robustness definition

Software robustness is a well known notion described as "the art of making software behave reasonably in exceptional situations". If the exceptional situations comes from the software itself (wrong execution paths), it leads to computational errors (division by zero). If it comes from the environment (wrong input values, memory alterations), it leads to functional errors (pressure is to high). It is clear that both notions interleave and should therefore be studied separately. The robustness is named intrinsic if it is about software failures, and extrinsic if it is about environment failures. The next sections recall the well known notion of intrinsic robustness and define the notion of extrinsic robustness.

A. Intrinsic robustness

A piece of software is intrinsically robust if it deals with all faulty paths within the source code. For example, an application that performs a division in statement *s* is robust with respect to the division by zero if the denominator is zero for none of the potential executions reaching *s*.

```
1  Assert (d != 0);     If (d != 0) /* error test*/
2  e = n/d;             { e = n/d;}
3                       else { /* error handler*/ };
```

Figure 1: Intrinsic robustness with respect to division by zero.

The enforcement of intrinsic robustness consists in the addition of error tests and handlers: a test branches either to the dangerous statement or the error handling statement. Figure 1 shows two sequences of statements robust with respect to the division by zero in statements `2`: in both examples the protection comes from statement `1`, it results in an execution stop in the left example, and the execution of an error handler statement `3` in the right example.

Intrinsic robustness can be automatically proven using static analyzers based on abstract interpretation [3] such as Astrée [1] or PolySpace [2]. Such tools can prove the source code safe with respect to dangerous constructs tagged as "unspecified, undefined or implementation defined" in the specification of the language ([6] for C90 or [5] for C++). They are sound and list all potential errors together with their locations in the source code: these results must be analyzed afterward to confirm whether potential errors are true errors or not.

B. Extrinsic robustness

Extrinsic robustness can be considered only after intrinsic one and deals with erroneous values coming from the environment. Note that we limit our study to un-intentional erroneous input values and memory alterations of the data only. As opposed to intrinsic robustness, there is no automatic tool designed to verify extrinsic robustness.

Erroneous values coming from the environment are either silly *global inputs* obtained from buggy devices (sensors, file system, other software), or data accessed after memory alterations. We consider the software memory as a data producer, and name *local inputs* all the memory accesses. Under this naming, a piece of software can be extrinsically robust if it trusts none of its global and local inputs. Until the end of this paper, robustness stands for "extrinsic robustness".

III. Robustness enforcement

To insure the robustness of a piece of software, the value of each un-trusted input must be checked against its correctness domain after its production and before its consumption.

```
   int Manage_Phase (int Phase,int Phase_move)
   {...
3  if ((Phase_move < 120) || (Phase_move > 170))
4  { /* handle the error */ }
5  /* else nothing to do */
   ...}
   {...
11 scanf(%n, Phase_id);
12 if ((Phase_id < 0) || (Phase_id >= MAX_PHASE))
13 { /* handle the phase identification error */ }
14 /* else nothing to do */
15 local = compute_phase_move(Phase_id);
16 Manage_Phase (Phase_id,local)
   ...}
```

Figure 2: Extrinsic robustness.

In practice, some lines of codes are added to test the input values and branch to an error handler if necessary. Figure 2 shows the validity check (lines `12-14`) for the global input `Phase_id` with respect to the specified domain `[0..MAX_PHASE]`: the global value must belong to the specified interval. The validity check of local input `Phase_move` is implemented lines `3-5`. This example demonstrates the complexity of robustness enforcement exemplified by three questions: should the value of variable `move` be checked, is the check domain `[120..170]` correct for local input `Phase_move` and is the check location `12-14` correct for global input `Phase_id`. Next sections describe succinctly the strategies chosen to circumvent these difficulties.

A. Un-trusted input

Considering all global inputs as un-trustable is practicable, but regarding all local inputs as un-trustable is not. We choose a subset of local inputs not to be trusted : the procedure inputs (parameters and global or static variables). This choice is based on two ideas: *(1)* the most dangerous program points with respect to memory errors are procedure calls because they imply the manipulation of a large amount of memory (all procedure parameters are put on the stack), and *(2)* the propagation of local errors must be stopped at the next call. As a result, this strategy protects the frontiers of the software: the un-trusted inputs are global and procedure inputs.

B. Correctness domains

Some correctness domains are straightforward: the user defined domains are used as correctness domains for all specified input. The un-specified global inputs are kept free and their correctness domains are full-range for their implementation type. The unspecified local correctness domains can only be computed from the software: we choose to compute them from correct values of the global inputs, by the execution of all the statements from the initial statement to the target statement. The computation of such domains is extremely difficult by hand because it requires the knowledge of the contribution of all execution paths, but it is done automatically by static analyzers.

C. Check locations

A correctness check for variable v must be performed before any use of the value of v. This seems to give a lot of freedom, but the TOCTOU (Time-Of-Check to Time-Of-Use [4]) paradigm reduces it drastically. To protect the program against memory alterations that could occur between the production (acquisition or computation) of the data and its consumption, the validity check must be performed "just before" the use. Then each input must be validated in each procedure that consumes it: this leads to a great number of checks.

IV. Semi automatic robustness verification

We have shown that robustness enforcement is a hard job but robustness verification is even worth. This task is performed manually in general, but we developed a semi-automatic verification method. We describe here the experimental results obtained by using a static tool. PolySpace offers one main functionality labeled *(F0)*: it proves the intrinsic correctness of the source code. We choose PolySpace to perform our experiment because it offers two other necessary functionalities: *(F1)* the computation and observation of numerical value domains, *(F2)* the evaluation of user level constraints. We have studied the correctness of an industrial piece of software implemented as 90 K-lines of C code associated to a specification of more than one hundred global inputs.

PolySpace has analyzed the original code after the correction of a few runtime errors *(F0)*. At this stage, the application was intrinsically correct. The global inputs have been constrained using the specification and the augmented source code has been automatically analyzed. This resulted in the detection of a handful of inconsistencies between the specification and the user constrains present in the source code *(F2)*. After correction of these inconsistencies, PolySpace computed the correctness domains for the local inputs *(F1)*. The robustness review of the industrial code has not yet been completed. It is done by manual code review to verify that all checks are present at right locations and that the checked domains are strictly included in the correctness domains computed automatically.

V. Conclusion

We have proved that it is possible to verify the robustness of a source code with respect to unintended spurious inputs and memory alterations in a partially automated way. Our industrial customer is satisfied by the current partial results and waits for the final ones. The local domains are of great help to improve the unit tests coverage.

We want to automate further the verification process by using the dead code detection *(F3)* offered by PolySpace. The idea is that: if a source code is intrinsically correct, the extrinsic error handlers should never be considered executable by a static tool. As for now, we have demonstrated a non standard usage of PolySpace: runtime errors detection gives preliminary results, but the value domains computation and constraint evaluation allow to go one step further and check for other properties.

The robustness property described in this paper is quite often used by industrials when their software needs frequent interactions with its environment in a safe context. The method developed for this study is general enough to be applied to any software. We have used one strategy to define untrusted local inputs and check locations, but other strategies should be studied. Given a strategy, the extrinsic robustness enforcement could be fully automated by code generation, and the robustness could be automatically verified.

References

[1] Astrée, `http://www.absint.com/astree/index_fr.htm`

[2] PolySpace, `http://www.mathworks.com/products/polyspace`

[3] *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.* Patrick Cousot & Radhia Cousot. In Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 238–252, Los Angeles, California, 1977. ACM Press, New York.

[4] *Secure programming using static analysis.* Brian Chess & Jacob West, 2007. Addison Wesley.

[5] Programming languages - C++. International standard ISO/EIC 14882: 1998 (E).

[6] Programming languages - C. International standard ISO/EIC 9899: 1990 (E).