*Robert Reitmeier*

Confidence in R2

Version: 1.0

Date: 2022-02-02

State: **in progress** / presented / released

Classification: Confidential

File: confidence\_in\_r2.docx

# History

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Change |
| 0.1 | 2022-01-28 | R. Reitmeier | Draft |
|  |  |  |  |

# Scope

This document gives a rationale why one can have high confidence using the dynamic analysis tool R2 for closing coverage gaps in C source code.

# Overview of the Core Functionality of R2

R2 is a dynamic analysis tool for C source code, implemented in the functional language Haskell. Based on the abstract syntax tree of C code, it performs a depth-first search over all control flow paths through a C function, including called subfunctions. All decision conditions (like “if”, “while”, “for”, conditional expressions etc.) on the way through the code are collected and a corresponding SMTLIB2 model is constructed. This model is passed on to a SMT solver in order to find solutions, which are then collected by R2. Finally, a report is created giving all stimuli for maximum code coverage, as well as all decisions that are considered dead code.

In addition, there are the following verification facilities provided by R2:

* Each stimulus that is found is verified by compiling and executing the source code, comparing the function’s predicted output value to the execution output value.
* By inserting a so called “solver\_find” pragma at specific location in the source code, it can be verified if the specific location was actually reached during execution of the code.
* By inserting a so called “solver\_debug” pragma at specific location in the source code, the theoretical prediction of an expression’s value at this location can be compared to the “real” execution runtime value. Currently, however, this has to be done manually if it is needed.

# Confidence in R2

## Use Case: Code Qualification

The use case of R2 is to find stimuli for C functions, so that subsequent test suite runs incorporating these stimuli give maximum reachable MC/DC code coverage. Unexplained gaps in code coverage lead to disqualification.

### Potential Errors

Refering to the description of the functionality of R2 (see chapter 2) and the code qualification use case, there are the following potential errors:

#### R2 gives a solution, but in reality, there is none.

This means that the stimuli for a specific path given by R2 does not make the control flow go along exactly this path during execution of the code on the target. The fact that there is no solution for this path does not matter here.

#### R2 does not give a solution, but there is one.

#### This corresponds to R2 claiming dead code, but it is not dead.

#### R2 gives a wrong solution.

This means that the found input stimuli for a specific path do not make the control flow go along the path during execution of the code on the target.

### Mitigations

For each of the above potential errors, we give a mitigation with a high detection probability (e.g. TD1 in ISO26262) as follows.

#### Mitigation for 3.1.1.1 “R2 gives a solution, but in reality, there is none.”

In our use case of code qualification, the test suite will be run with the relevant stimuli found by R2. Also, the code coverage of the test run will be measured in order to fulfill safety requirements. Assuming a trustworthy code coverage measurement tool, the coverage gap left behind by this stimulus would be found (there cannot be another stimulus that could cover the current path, since by assumption, there is no such stimulus). Hence, this potential error would be detected with high probability.

#### Mitigation for 3.1.1.2 “R2 does not give a solution, but there is one.”

This means that R2 reports a certain location on the path as dead code, but in fact there would be a stimulus that would tread the path. If no other stimulus in the test suite covers this path by chance, there is a safety concern because there would be some live code left untested (“undead code”).

The question if R2 delivers a stimulus for a path boils down to the correctness of the formula that is created by R2. Correctness in this case means that it accurately models the all calculations happening on the path as they would happen executing the compiled code.

If R2 does not give a stimulus for a path, although there is one, this means that either

1. the formula derived by R2 is incorrect or
2. the subsequent SMT solver did not find the solution.

Let us look at these two possibilites in the next subchapters.

##### The formula is incorrect

The formulae produced by R2 as inputs to the SMT solver consist of a number of variable declarations and assertions.

Having a incorrect formula leaves us with the follwing possible reasons for this (or any combination of these):

1. A variable declaration is missing  
   → the SMT solver would complain about an undeclared variable.
2. Too many variable declarations  
   → either an additional variable declaration redeclares the same variable, then the SMT solver would complain, or the additional declaration is simply superfluous and harmless.
3. Wrong variable declaration  
   → this would either lead to a
   1. type error in the formula when combining the variable with other variables in expressions, which would be noticed, or,
   2. if the types are consistently wrong up to the function’s arguments and down to the function’s return value, the user would notice wrong argument or return value types.  
      Hence we have the check “User checks the argument and return value types in the formula for reportedly dead code”.
   3. If the types are not consistently wrong and there is an implict cast undoing and therefore shadowing the typing error, the function’s return value must have been affected (if the typing error had impact). In this case, we can detect the affected return value with the solution check that is always performed for any solution.
4. An assertion is missing.  
   Adding more assertions does only even more restrict a formula which is incorrect already. A formula with more assertions does not produce more solutions (monotonicity of the logical conjunction), hence this case is not a possible reason for the formula without the missing assertions incorrectly giving no solutions.
5. There are too many assertions.  
   Each assertion in the formula stems from a decision in the code (maybe in more than one instance, in case of a loop condition, e.g.). The number of assertions in the formula can easily be reviewed. Hence we have the  
   **CHECK “If R2 reports dead code, manually verify the number of assertions in the formula against the number of decisions on the path in the code.”**
6. An assertion is wrong.  
   Following the assumption in 3.1.1.2 that R2 reports the code snippet as dead, consider the other branch(es) that are not dead (if all are dead, go to the uppermost decision that is partly dead. This induction stops at the top level decision, since obvisouly this can’t be fully dead). The assertions for the other branches consist of negations from the assertion in question. If all the other paths have been shown as live and the solution of all other paths was verified against the actual compiled code return value, the probability is high that the assertion is correct (although not 100%, of course).
7. The maximum loop unrolling threshold is too low, meaning that a loop was not unrolled often enough so that the live path is detected. In this case, increase the loop unrolling bound to a value that is still acceptable from a run time perspective. We are making use of the bounded model checking heuristics, which states that all errors in deeper nestings of the system can also occur in shallow nestings. But this is just a heuristic, leaving a residual risk. However, we consider this residual risk as low.
8. The loop length prediction is incorrect. For reportedly dead code, manually confirm the inferred loop length or switch off loop length prediction.

Moreover, one can partly confirm that reportedly dead code is really dead by running a brute force (or maybe a more clever i.e. restricted) search with the compiled code, triggering an alert when “undead” code is hit. This method assumes that solutions for seldomly trodden paths are not unique in the whole search space, but somewhat evenly distributed, so that we can find solutions by only searching a small contiguous fraction of the whole exploded search space.

##### SMT solver did not find the solution

In every case where R2 reports dead code, a human being that is capable of this task will analyze and review the code snippet in question, trying to cover the presumably dead code manually. For this, both low level calculations and high level argumentation can be used.

Since this is human activity, it cannot be assigned a tool detection level.

#### Mitigation for 3.1.1.3 “R2 gives a wrong solution.”

We can refer to 3.1.2.1 here, with one restriction: There might be another stimulus in the test suite, that covers the current path. However, in this case, the current path is then covered by this other stimulus, and there is no safety concern since the current path is covered anyway, no matter by which stimulus (no impact).

# Summary

We have enumerated all potential errors of R2, together with mitigations where the potential errors actually have impact. All mitigations have high detection probability, therefore we can consider R2 as TCL1 in ISO26262 context.