# Verification-Cases: Characterizing the Completeness Degree of Incomplete Verification for C Programs

## Towards Using Formal Verification for Low Criticality Functions

Daniel Ratiu
Siemens Corporate Technology
Research and Technology Center
Otto-Hahn-Ring 6
81739 Munich, Germany
daniel.ratiu@siemens.com

Vincent Nimal
Department of Computer Science
University of Oxford
Oxford OX1 3QD, UK
vincent.nimal@cs.ox.ac.uk

## ABSTRACT

Current code-level verification tools are powerful enough to be usable in industry for performing functional verification at sub-system level. Due to the nature of these tools (e.g. bounded model checkers) or to abstractions that are performed in order to make the verification possible (e.g. assumptions about the environment of the system or the used APIs), the verification results are not complete: some errors may remain undetected. Even in this case, the verification results can be good enough for verifying functionality at low criticality levels. Furthermore, when the sources of incompleteness are well understood, the results can be combined with testing for verifying highly critical code. In order to use these results in building assurance cases, we need systematic means to characterize the completeness degree of incomplete verification. In this paper we introduce the concept of a "*verification case*" as means to make explicit the incomplete parts of verification. We characterize common sources of incompleteness when verifying C programs and present our tool support for characterizing incompleteness of verification when analyzing C code using the CBMC model checker. This paper is part of our efforts to use language engineering technologies in order to make formal verification accessible for practicing engineers.

## Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Software Quality Assurance; D.2.4 [**Software/Program Verification**]: Model Checking

## 1. INTRODUCTION

In the last decade formal verification techniques at code level have become mature enough to be applied in practice. Numerous papers [1, 2, 6, 9, 11, 14] investigate the use of C model checking techniques to find bugs in large systems. Some [12, 16, 18] even analyze fragments of the Linux kernel, demonstrating the feasibility of verification for "real world" programs. Formal methods are used by industrial actors from critical sectors like aviation and nuclear. However, the status quo w.r.t. the use of verification in "common" industry is significantly behind what the technology allows.

Even in the development of safety critical systems, formal verification techniques are not broadly used. Apart from the cases in which the use of formal methods is highly recommended by safety standards like IEC 61508 (for SIL 4), developers are reluctant to use verification in the development of functions with lower criticality (SIL 1, SIL 2 or even SIL 3). The lack of affordable, qualified, robust tools, some misconceptions regarding the inherent cost of formal verification and the complexity of these techniques are often the justifications to the absence of formal verification in a low criticality project.

Using formal verification is challenging for non-experts. Furthermore, the model checking tools available for C programs often face important scalability issues. Achieving a *complete* formal verification in this context is therefore a very complex task, many times in infeasible. Incomplete verification is, however, much easier to perform in practice on a continuous basis. We conjecture that incomplete functional verification can be economically feasible, also in common projects involving low critical software functions. *The most important obstacle to overcome is that it is currently unclear how the results of incomplete verification can be used as evidence in assurance arguments for certification.*

*Working context: mbeddr and CBMC.* Our long term work is focused on drastically improving the usability of formal verification tools up to the point where they are usable by practicing engineers on a daily basis. Our approach relies on domain specific languages, in order to mask the complexity of working directly with verification tools. We have built a stack of C code and domain specific analyses as part of the mbeddr[1] project [15, 17]. *mbeddr aims to simplify the verification process from the point of view of the end-users, with the objective of making formal verification as easy to use as testing.* All of the following examples are done using

[1]`http://www.mbeddr.com`

mbeddr and CBMC. We are, however, convinced that the results presented here can be easily generalized and applied to other tools.

*Testing vs. verifying.* Writing test-cases represent the most widespread manner today to verify functional requirements at subsystem level in the industry. Test-cases are relatively easy to understand, use and review by embedded systems practitioners. Furthermore, running many test-cases is very fast, developers get quick feedback about the results and when tests fail they are usually easy to debug. Measuring the quality of test-cases is done through code-level coverage tools. For SIL 1 and SIL 2 functions, for example, IEC 61508 requires only a statement coverage of the tests, basically that for each safety requirement there are enough tests such that each statement is executed at least once. For SIL 3 functions, the standard requires a condition coverage. From the point of view of the verified requirement, starting from SIL 2, IEC 61508 highly recommends that the tests consider equivalence classes and input partition testing, including boundary value analysis.

Yet, a suite of tests covers only points from the state space of the system as illustrated in Fig. 1. Tests are finite by nature, whereas software with unbounded loops are not uncommon – e.g., for controls.

Verifying a system allows us to reason over several paths at the same time and cover a set of initial values with a single run of a model checker. In Fig. 1, the regions delimited by the blue box and the red boxes represent the state space analyzed – covering the same space with tests may require a very large number of test vectors.

Even though model checking would cover a state space larger than the one covered by some tests, these tests would often be preferred, as they correspond to a concrete reality whose bounds and assumptions are visible to the practitioners. Additionally, after the verification is initially performed by an engineer, it will be applied again and again as new changes are incorporated. Therefore, the verification needs to be understood by others and ultimately by an external assessor in order to provide certification.

*Structure and contributions.* In this work, we aim to characterize the assurance level that can be given by incomplete verification. When incomplete verification is used, the engineers need to assess the limitations of the analysis, or, in other words, to assess its completeness degree. In Sec. 2, we describe typical sources of incompleteness when using code-level model-checking. The set of assumptions making the verification incomplete needs to be characterizable in an explicit manner, so that the confidence in the verification results can be quantified. In Sec. 3, we develop the concept of *verification-case* in order to emphasizes on the incomplete nature of verification. In Sec. 4, we give an overview over the related work and then we conclude the paper and present our future plans in Sec. 5.

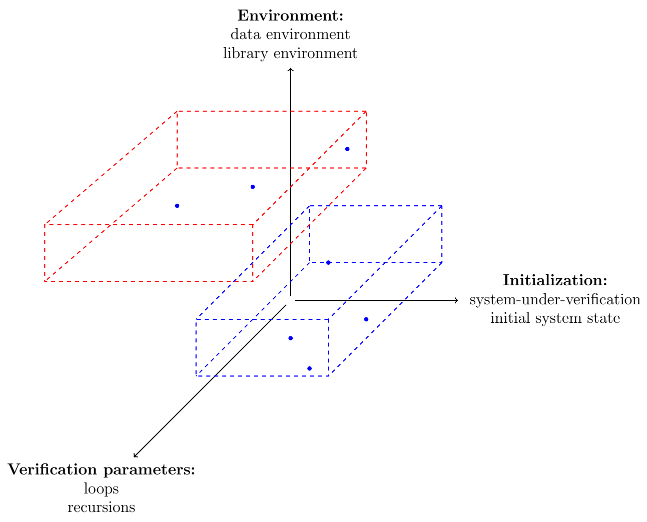


**Figure 1: Test-cases vs. verification-cases.**

## 2. INCOMPLETENESS OF VERIFICATION

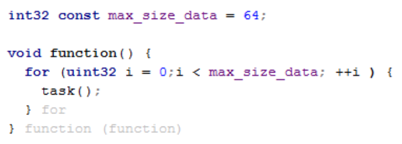
Even though verification covers a larger portion of the state space than testing does, it will still remain incomplete in many practical cases. Each of the following sub-sections presents an aspect of this incompleteness along with a small example. These aspects are orthogonal, so a particular verification can provide results that are incomplete along several dimensions at the same time.

### 2.1 Initialization

The verification of a system needs to start from an *initial* state space. This covers in particular the initial values of global and static structures and variables, the values of the parameters passed as arguments to the system-under-verification and the objects in memory (e.g., a chained list in the heap).

*Choosing configuration parameters.* Many embedded software systems are parametrized by constants used for calibration or different configurations of the code. The values of these parameters have a highly restricted range and known in advance. The set of possible valid values is often discrete. Typically, these parameters are flashed on a read-only memory before the product is shipped or during an off-line maintenance. The parameters are subsequently used as constants during the run of the code.

In Fig. 5, the maximum size of the data treated is fixed in the code as a global constant.



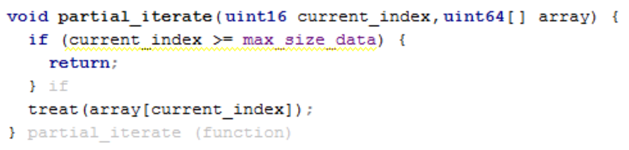
```
int32 const max_size_data = 64;

void function() {
  for (uint32 i = 0;i < max_size_data; ++i ) {
    task();
  } for
} function (function)
```

**Figure 2: Example of configuration parameter: `max_size_data`.**

*Choosing the initial state.* Many times, the system-under-verification will run into a loop and do some computations iteratively (e.g., read sensors values, then perform computation, then issue commands). Often such systems have an internal state which can be reached from the program start after a high number of iterations. We can, however, let the verification start from an arbitrary state by (nondeterministically) choosing values for the state variables. By choosing the initial state in a nondeterministic manner the initial state that we chose might be a subset of all possible states that the system-under-verification could reach during a run and thereby the assumption for our verification is too narrow and we cover only a subset of the possible reachable states.

In Fig. 3, the function `partial_iterate` is intended to be called in a loop for each element of an array (passed as argument). If a value unexpected by the treatment function `treat` is in `array[65535]`, we need to iterate 65535 before finding this value. If we choose nondeterministically the index used to execute this function, we will reach directly the case in which the unexpected value is read by `treat`.



```
void partial_iterate(uint16 current_index,uint64[] array) {
  if (current_index >= max_size_data) {
    return;
  } if
  treat(array[current_index]);
} partial_iterate (function)
```

**Figure 3: Example of initial state: `current_index`.**

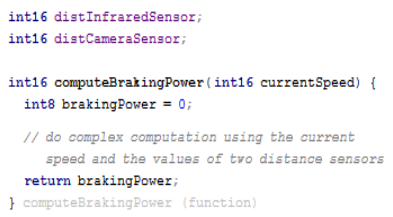
## 2.2 Environment

Once the system-under-verification has been defined (typically a program module called by a unique function), the environment represents all the other parts of the system that interact with the system-under-verification. This covers, e.g., global variables, files and calls to APIs.

*Data environment.* The inputs of the system are provided by the environment. In order to make the verification feasible, we typically restrict the input space. A good example for this is the case of embedded programs that process information from sensors. We might assume – for the verification – that the data provided by the sensors lie within a certain range. Another example are cases where the system-under-verification manipulates recursive data structures or arrays of variable length. In these cases, we might choose to set a bound on their sizes.

For example, in Fig. 4, we present a function which computes the braking power. This function uses the `currentSpeed` and the value of two distance sensors. We can perform the verification for example by covering only a subspace of the possible inputs like speed is between [0..50] meter per second and distance between [0..200] meters.

*Library environment.* In some cases, the system-under-verification makes use of libraries which provide lower-level services (e.g., calls to standard libraries or device drivers APIs). The implementation of these libraries might be unavailable when the code is analyzed, or might be too complex



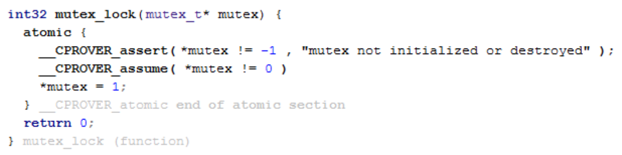
```
int16 distInfraredSensor;
int16 distCameraSensor;

int16 computeBrakingPower(int16 currentSpeed) {
  int8 brakingPower = 0;
  // do complex computation using the current
     speed and the values of two distance sensors
  return brakingPower;
} computeBrakingPower (function)
```

**Figure 4: Example of data environment: `distInfraredSensor`, `distCameraSensor`.**

to analyze in addition to the system-under-verification. In these cases, mock-ups or models of these libraries can be used instead; this leads however to incomplete results.

In Fig. 5, the implementation of a mutex lock is not provided. Assuming that the function is correctly implemented, we abstract this function by an equivalent model. In detail, this function atomically checks if the mutex has correctly been initialized. If it is the case, it then ensures that after the assumption, the mutex is released so that `mutex_lock` can acquire it. In essence, this assumption could replace a blocking while which would spin until the lock was released by another thread. `mutex_lock` acquires the lock at the end.



```
int32 mutex_lock(mutex_t* mutex) {
  atomic {
    __CPROVER_assert( *mutex != -1 , "mutex not initialized or destroyed" );
    __CPROVER_assume( *mutex != 0 )
    *mutex = 1;
  } __CPROVER_atomic end of atomic section
  return 0;
} mutex_lock (function)
```

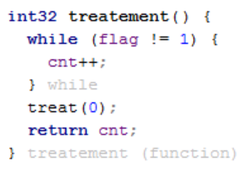
**Figure 5: Example of library environment: `mutex_lock`.**

## 2.3 Verification parameters

The exploration of the state space by the model checker may be incomplete by itself. Model checking in general is known to be undecidable – the analysis may never terminate. Incomplete verification tools thus introduce some parameters to limit the exploration. Bounded model checkers impose, for instance, to set loop and recursion unwinding bounds. These restrictions restore the decidability of the analysis, at the cost of an additional incompleteness degree.

*Loops unwinding.* Bounded model checkers must unwind the loops in order to perform verification. Very often, the complete unwinding cannot be performed (eg, the loop itself depends on a variable input parameter, the loop is unbounded and the breaking condition is complex, or simply very deep unwinding makes the verification too complex). In such cases, engineers need to decide on the depth of the unwinding and this leads again to incomplete results.

In Fig. 6, the function `treatment` waits in a blocking while until another process set the volatile global variable `flag` to 1. It then calls the function `treat`. During the blocking while, it counts the number of iterations in `cnt`, which is a `uint32`. If the engineers want to observe the overflow of `cnt`, they need to unwind at least 65535 times the while loop. If `cnt` is not the center of their attention, they can unwind an arbitrary number of times. In the case of blocking loops, they can also abstract the loop by replacing it with an assumption imposing its final state – here, `assume(flag == 1)` – at the cost of abstracting completely `cnt`.

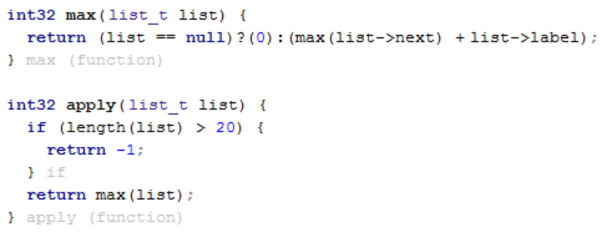


```
int32 treatement() {
  while (flag != 1) {
    cnt++;
  } while
  treat(0);
  return cnt;
} treatement (function)
```

**Figure 6: Example of loop unwinding: blocking while.**

*Recursion bounds.* Similarly to loops, recursive calls can appear in the code. According to IEC 61508, a limited use of recursion is allowed for SIL 1 and SIL 2 functions. Other coding standards like MISRA forbid the use of recursion. In comparison to loops, recursions is often more problematic, since each recursive call consumes memory on the stack (depending on the amount of data passed between calls and local variables). When performing the verification using bounded model checkers at C level, engineers need to decide on the depth of unwinding of the recursive calls making thereby the verification incomplete

In Fig. 7, the recursive function `max` is recursively called on all the elements of a list of an arbitrary size. Any unwinding of this function would thus be partial. However, this function is called in the specific context of `apply`, which limits the number of elements to 20. In this case, the complete verification of `max` called in the context of `apply` only can be performed with unwinding 20 times the recursion.



```
int32 max(list_t list) {
  return (list == null)?(0):(max(list->next) + list->label);
} max (function)

int32 apply(list_t list) {
  if (length(list) > 20) {
    return -1;
  } if
  return max(list);
} apply (function)
```

**Figure 7: Example of recursion unwinding: `max`.**

## 3. VERIFICATION CASES

A *verification-case* combines the information about the system-under-verification, the environment definition, the initial state of the system and the parameters with which the verification tool is called. Thereby it makes explicit all the assumptions that are done when the verification is performed.

In Fig. 8, we present an overview of our approach. As a first step, the engineers define a verification-case for a system. Based on the system definition and on the verification-case, mbeddr will generate C code which is subsequently fed as input into CBMC (the verification tool that we use). CBMC will perform the actual verification using the parameters specified in the verification-case and return a set of results. These results can then be used as evidence in an assurance case together with the *verification-case* definition – which characterizes the completeness degree of verification and thus the confidence upon the results. In the following, we will describe in more detail the process of building the verification-case and using the verification results as evidence.

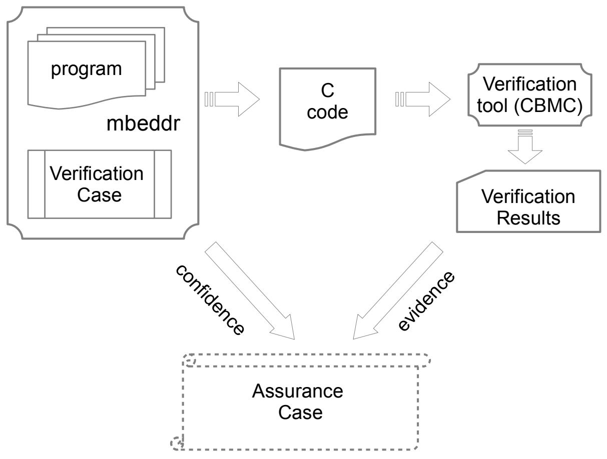


**Figure 8: Overview of our approach.**

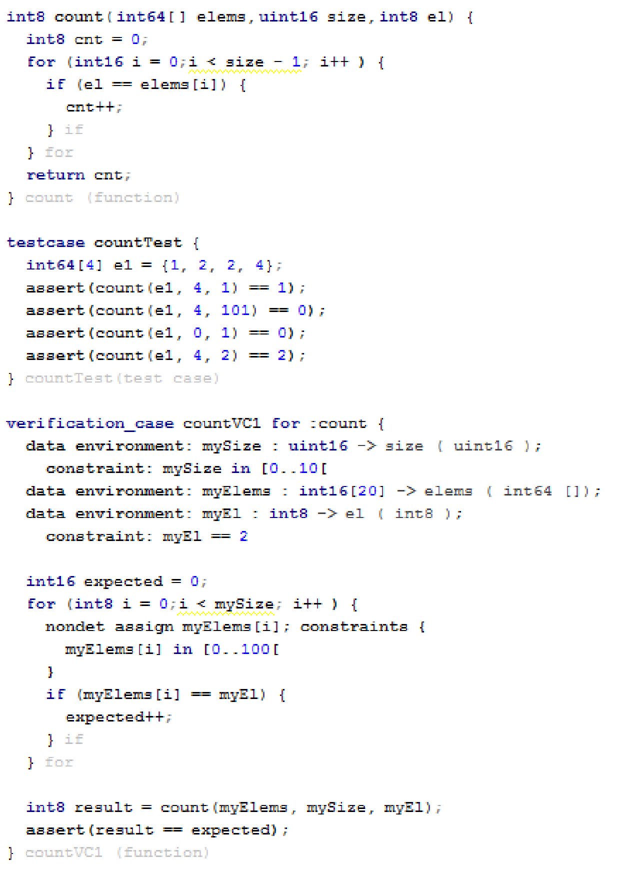
### 3.1 Defining a verification-case

mbeddr provides a first-class language construct to define verification cases. In order to verify a piece of code in mbeddr, embedded software engineers need to build a verification-case and perform the following steps:

1. *Delimit the system* under verification (at C level, this is usually one or more C functions)
2. *Define the initial state* from which the verification will be performed (eg, initialize global variables).
3. *Define the environment* in which the system-under-verification will run. The environment definition captures the input data to the system. If the system-under-verification is part of a layered architecture and makes use of lower-level functionalities, developers must decide where they integrate these functionalities as part of the system-under-verification or not.
4. *Choose verification parameters* with which CBMC will be called. Important parameters that affect completeness of the verification results are, for example, the loops unwinding.
5. *Define the property to be verified* by using C level assertions. This property can match exactly the requirement specification or can be a restriction thereof.

In the upper part of Fig. 9, we present a simple C function which takes an array of integers (`elems`) and an element to be searched in this array (`el`). The function returns the number of occurrence of the element in the array. In the lower part of Fig. 9, we present a test-case that checks if the function exhibits the desired behavior. This test-case is sufficient to achieve a condition coverage at C level. In the lower part of the figure we present a “verification-case” which defines the environment for performing the verification, initializes the input variables and checks the expected behavior. The data belonging to the environment is highly restricted when compared to the variable space that the `count` functions accepts: the size of the input vector (`mySize`) is between 0 and 10, the elements of the vector are between 0 and 100 even if the `count` function takes a broader input (int64), and the value of the searched element is fixed.

By looking at the implementation of the `count` function, we notice two bugs. The first bug is that the for-loop does not iterate over the entire array and therefore the occurrence of the searched element at the last index would not be counted. The second bug is that if elements array is very large and the searched element occurs many times, the variable `cnt` will overflow. Our tests example misses both of these bugs. However, the verification (even if incomplete) catches the first bug but misses the second one.



```
int8 count(int64[] elems,uint16 size,int8 el) {
  int8 cnt = 0;
  for (int16 i = 0;i < size - 1; i++ ) {
    if (el == elems[i]) {
      cnt++;
    } if
  } for
  return cnt;
} count (function)

testcase countTest {
  int64[4] e1 = {1, 2, 2, 4};
  assert(count(e1, 4, 1) == 1);
  assert(count(e1, 4, 101) == 0);
  assert(count(e1, 0, 1) == 0);
  assert(count(e1, 4, 2) == 2);
} countTest(test case)

verification_case countVC1 for :count {
  data environment: mySize : uint16 -> size ( uint16 );
    constraint: mySize in [0..10[
  data environment: myElems : int16[20] -> elems ( int64 []);
  data environment: myEl : int8 -> el ( int8 );
    constraint: myEl == 2

  int16 expected = 0;
  for (int8 i = 0;i < mySize; i++ ) {
    nondet assign myElems[i]; constraints {
      myElems[i] in [0..100[
    }
    if (myElems[i] == myEl) {
      expected++;
    } if
  } for

  int8 result = count(myElems, mySize, myEl);
  assert(result == expected);
} countVC1 (function)
```

**Figure 9: Motivating example: simple function to be verified, a test case and s verification case.**

## 3.2 Confidence in incomplete results

If the verification is successful (i.e. the property representing requirements holds), the results can be used as evidence in an assurance case. Verification-cases allow engineers to make explicit all assumptions and limitations of the verification that can cause incomplete results. Each of the sources of incompleteness should be explicitly treated and additional assurance might be necessary.

*Incompleteness due to the chosen initial state.* When the choice of the initial state for the verification is too restricted, the verification might miss the property violation that would occur for some other uncovered, yet possible initial states. The completeness degree can be increased by defining additional verification-cases specifying other initial states. When the choice of initial states is to broad, it might cover impossible initial states. Furthermore, some of the “valid initial states”, albeit possible, could be reached by the system only after a very long run. It is highly improbable that such states would be reach in the production system.

*Incompleteness due to the chosen environment.* Choosing only small regions from the possible input space or using abstractions for the API calls can significantly speedup the verification. This generates incomplete results, since the property under check might be violated by verification with inputs that are not considered. In these cases, verification can be complemented with tests which can provide additional confidence for the evidence.

*Incompleteness due to unwinding bounds.* Limiting the unwinding of loops is often unavoidable when using bounded model checkers. However, the chosen unwinding can be large enough to produce verification results with high coverage. Even if the unwinding is incomplete (e.g., in case of unbounded loops), the verification tool can still provide program traces that were explored. These traces can subsequently be used to measure the coverage of the verification at code level. Note that the model checker that we use can also unwind as many time as required if the bound is obvious (e.g., in for-loop with fixed values of the form `for(i = 0; i<N; i++)`).

## 4. RELATED WORK

*Making the verification incompleteness explicit.* To the best of our knowledge, making the incompleteness of incomplete verification explicit was not formally addressed in the literature.

*Verfication for software assurance.* In [5], Coppit et al. use bounded exhaustive testing to provide software assurance. Similarly to our approach, but in the context of tests, they explicitly state the limits of these tests and the confidence that can be obtained out of their results. Shen and Shapiro explore in [3] the application of model checking to software assurance. In particular, they state precisely under which conditions the tool is sound. They tell their experience of using (incomplete) model checking in building confidence.

*Interpretation of specific bounds in incomplete verification.* Often, authors specialize in one or two specific incomplete parameters of their approach. For example, Kroening et al. [4] explicitly mention the boundedness of the state space exploration: the tool CBMC will only prove program for the number of loop unwinding provided. An assertion is nevertheless inserted at the end of the loop unwindings, so that simple cases where the state space can exhaustively be explored after a few unwindings can be covered.

In the domain of concurrency, Donaldson et al. use a maximum number of threads [7]. Lal and Reps use alternatively a maximum number of context-switches [13]. Joshi and Kroening suggest to limit the number of weak memory reorderings allowed in a counter-example [10]. Many other measures can be found in the verification literature.

*Test-cases for verification.* Verification-cases are analogous to test-cases. This observation was already empirically made by some developers. Rapicault already implemented verification tasks as unit tests in the Eclipse plugin of CBMC[2]. A similar approach has been explored by Nimal in an improvement of the Visual Studio plugin of CBMC[3]. Yet, no interpretation of these verification tasks is provided.

## 5. CONCLUSION AND FUTURE WORK

In this paper, we presented a first attempt to qualify explicitly the completeness degree of incomplete verification. To the best of our knowledge, this characterization had not been formulated in the past. We introduced the verification-case, which is analogous to a test-case but covers a larger region of the state space. We instantiated in mbeddr verification-cases for CBMC, with limitations specific to bounded model checking. We hope that the verification-cases will mark a first step towards building assurance cases using incomplete verification results.

There are additional parameters that are currently not taken into account in these verification-cases. The platform and operating systems on which results are valid, the processor architectures, the memory management are elements that also need to appear in an additional dimension of our verification space in Fig. 1. Parameters like the thread bounds, the maximum number of thread interferences or interrupt priorities should also complete the verification-case component "Verification parameters". We leave this as future work. Other measures introduced by the verification community like the maximum number of context-switches or the number of weak memory reorderings in a trace would also be relevant to verification-cases, but might be correlated to other existing components.

We addressed in this paper the case of low critical systems. Incomplete verification can also be used for highly-critical systems: if we understand the state space coverage of a verification-case (like the boxes in Fig. 1), we can combine testing and verification. Quantitative evaluation of the coverage of a verification-case remains a challenge.

## 6. REFERENCES

[2]`http://www.cprover.org/eclipse-plugin/`

[3]This plugin has not been released yet.

[1] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with slam. *Commun. ACM*, 54(7):68–76, July 2011.

[2] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker blast. *STTT*, 9(5-6):505–525, 2007.

[3] H. Chen and J. S. Shapiro. Exploring static checking for software assurance. Technical Report SRL-2003-06, SRL, CS, Johns Hopkins University, 2003.

[4] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, pages 168–176, 2004.

[5] D. Coppit, J. Yang, S. Khurshid, W. Le, and K. Sullivan. Software assurance by bounded exhaustive testing. *Software Engineering, IEEE Transactions on*, 31(4):328–339, April 2005.

[6] L. C. Cordeiro, B. Fischer, and J. Marques-Silva. Smt-based bounded model checking for embedded ANSI-C software. *IEEE Trans. Software Eng.*, 38(4):957–974, 2012.

[7] A. F. Donaldson, A. Kaiser, D. Kroening, and T. Wahl. Symmetry-aware predicate abstraction for shared-variable concurrent programs. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 356–371, 2011.

[8] A. F. Donaldson, D. Kroening, and P. Rümmer. Automatic analysis of DMA races using model checking and $k$-induction. *Formal Methods in System Design*, 39(1):83–113, 2011.

[9] J. Franklin, S. Chaki, A. Datta, J. M. McCune, and A. Vasudevan. Parametric verification of address space separation. In *Principles of Security and Trust - First International Conference, POST 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012, Proceedings*, pages 51–68, 2012.

[10] S. Joshi and D. Kroening. Property-driven fence insertion using reorder bounded model checking. In *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*, pages 291–307, 2015.

[11] M. Kim, Y. Kim, and H. Kim. A comparative study of software model checkers as unit testing tools: An industrial case study. *IEEE Trans. Software Eng.*, 37(2):146–160, 2011.

[12] D. Kroening and M. Tautschnig. Automating software analysis at large scale. In *Mathematical and Engineering Methods in Computer Science*, volume 8934 of *LNCS*, pages 30–39. Springer, 2014. invited paper.

[13] A. Lal and T. W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1):73–97, 2009.

[14] L. Pike, S. Niller, and N. Wegmann. Runtime verification for ultra-critical systems. In *Runtime*

*Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*, pages 310–324, 2011.

[15] D. Ratiu, M. Voelter, B. Kolb, and B. Schätz. Using language engineering to lift languages and analyses at the domain level. In *NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings*, pages 465–471, 2013.

[16] B. Schwarz, H. Chen, D. Wagner, G. Morrison, J. West, J. Lin, and W. Tu. Model checking an entire linux distribution for security violations. In *Computer Security Applications Conference, 21st Annual*, pages 10 pp.–22, Dec 2005.

[17] M. Voelter, D. Ratiu, B. Kolb, and B. Schätz. mbeddr: instantiating a language workbench in the embedded software domain. *Autom. Softw. Eng.*, 20(3):339–390, 2013.

[18] T. Witkowski, N. Blanc, D. Kroening, and G. Weissenbacher. Model checking concurrent linux device drivers. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, pages 501–504, 2007.