On Using Results of Code-level Bounded Model Checking in Assurance Cases

Carmen Cârlan, Daniel Ratiu, and Bernhard Schätz

fortiss GmbH, Munich, email: carlan@fortiss.org
Siemens CT, Munich, email: daniel.ratiu@siemens.com
fortiss GmbH, Munich, email: schaetz@fortiss.org

Abstract. Software bounded model checkers (BMC) are today powerful tools to perform verification at unit level, but are not used at their potential in the safety critical context. One reason for this is that model checkers often provide only incomplete results when used on real code due to restrictions placed on the environment of the system in order to facilitate the verification. In order to use these results as evidence in an assurance case, one needs to characterize the incompleteness and mitigate the assurance deficits. In this paper we present an assurance case pattern which addresses the disciplined use of successful but possibly incomplete verification results obtained through C-level bounded model checking as evidence in certification. We propose a strategy to express the confidence in incomplete verification results by complementing them with classical testing, and to mitigate the assurance deficits with additional tests. We present our preliminary experience with using the CBMC model checker and the mbeddr environment to verify three safety-critical software components.

Keywords: assurance cases, bounded model checking, confidence arguments

1 Introduction

Modern software model checkers are powerful enough to verify complex properties of programs at unit level. In the field of safety critical systems development, formal verification is used only for highest critical functions and when it is highly recommended by safety standards like IEC 61508 [2]. Instead, current functional verification of software is mostly based on testing.

Figure 1 presents three fragments of an assurance case (in a Goal Structuring Notation-like notation [3]) for the correct implementation of a safety requirement by a software component. In test-based verification (Figure 1-left-up), the assurance of the correctness of the developed software is split into two parts: the *conformance of the implemented behavior* with the test-suite demonstrating the *validity* of the correctness claim with respect to the selected test case, and the *analysis of the coverage of the implemented behavior* by the selected test-cases demonstrating the *confidence* in the correctness claim. The required

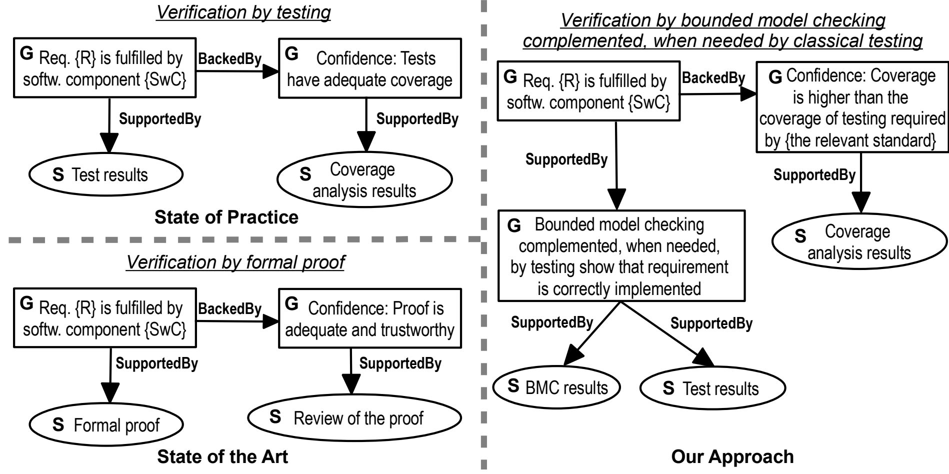

Fig. 1. Existing approaches for providing evidence in assurance cases: testing or formal proofs (left); Our approach proposes to combine model checking with testing (right).

coverage grows with the assurance level, in case of the IEC 61508 from statement via branch to MC/DC. In the case of verification by formal proofs (Figure 1-left-down), one argues the *confidence* in the results by claiming adequacy and trustworthiness of the proof as a demonstration that SwC fulfills R.

The idea of splitting the argumentation in a part focusing on the *conformance of the implemented behavior with the requirements*, and a part focusing on the *confidence* [14] can also be applied in the verification which uses bounded model checking (Figure 1-right). In this case, we split the argumentation into the proof of the correctness of the implemented behavior with respect to a specification under restricting assumptions, and the demonstration of sufficient confidence in the respective restricting assumptions. If the confidence argument is not strong enough, we propose to use classical testing for compensating the identified assurance deficits.

In this paper, we present an assurance case pattern which can be used at the interface between developers, verification engineers, safety managers and third party assessors to tackle the following questions: **Q1)** *How can we use successful verification results of software bounded model checkers as evidence for the correctness of the implementation of software components?* **Q2)** *How can we cover the assurance deficits due to incomplete verification using classical testing?* This work is part of our efforts at fortiss GmbH and Siemens to enable practicing engineers to use successful results of code level bounded model checking as evidence for certification.

Contributions. We present a pattern to use successful, but possibly incomplete bounded model checking verification results as evidence in assurance cases. In case these verification results are incomplete, we develop a confidence argument by comparing the input and the state coverage of incomplete model-checking

with coverage requirements for classical testing. We present our experience with using bounded model checking on three real-world safety-critical software components.

Structure of this paper. In Section 2, we present an assurance case pattern to incorporate results of the bounded model checker-based verification as evidence for assurance (Q1). In Section 3, we characterize the confidence in incomplete verification results and the additional testing in an argument structure pattern (Q2). In Section 4, we present our experience with verifying three software components. In Section 5, we discuss variability points of our approach. The last two sections contain the related work and conclusions.

2 Using BMC Results as Formal Verification Evidence

Testing is the state-of-the-practice verification method. However, safety standards recommend the usage of formal verification results as evidence for certification, because formal verification allows exploration of all possible behaviors while assessing the satisfaction of a certain safety property. When complete verification is not possible, standards require that the limits of the coverage of the performed verification are explicitly expressed. If bounded model checking is used (as alternative verification method), DO-178C recommends the construction of an assurance case in order to argue the adequacy and trustworthiness of the verification results for demonstrating that safety goals have been met. In the following, we develop an assurance case pattern for arguing that the objective related to the functional correctness of a software module has been met by bounded model checking accompanied by testing, when needed.

System under verification. Our focus is on code-level functional verification of reactive software components. A software component (SwC) possesses an internal state, a set of input variables with different types $I = \{i_1 : T_1, ..., i_n : T_m\}$ and a set of output variables. Each type T_l defines a set of possible values which can be taken by an input variable. Being a reactive system, the component is called in a (possibly infinite) main loop. For each of the steps of the loop, each of these input variables can take a different value – let i_l^t denote the value of an input variable i_l at time step t. The value i_l^t conforms to the type of i_l, namely $i_l^t \in T_l$.

Main Pattern. Figure 2 presents a pattern that captures the structure of an assurance argument, which uses as evidence bounded model checking results together with classical testing, if the verification is incomplete. Our top-level goal $G1$ is that a software component SwC implements a safety requirement formalized as a property P, given the environmental constraints Env^{SwC} ($C0$). Env^{SwC} assigns to each input variable its step-dependent range: $Env^{SwC}(i_l)(t) \subseteq T_l$.

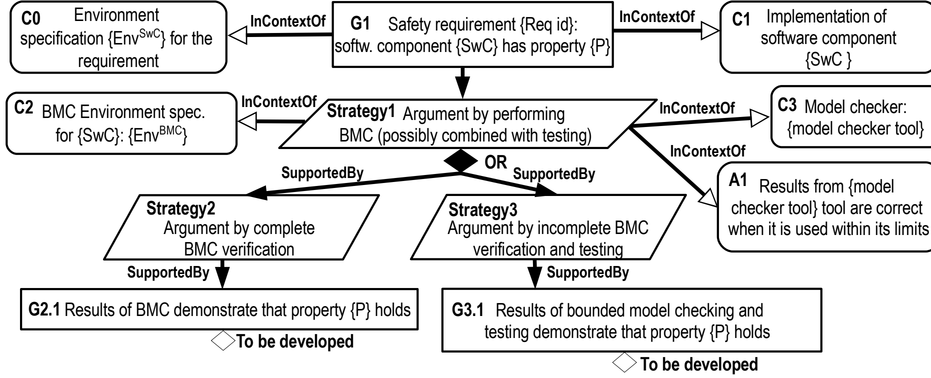

Fig. 2. Main pattern for arguing that a software component implements a safety requirement. The argument's strategy is to use results from bounded model checking verification complemented, when necessary, by testing.

Strategy 1: Argument by combining BMC with testing. Our strategy to decompose the top-level goal is to use bounded model checking on the source code of the component ($C1$), possibly combined, when needed, with testing. The bounded model checker uses the environment definition Env^{BMC} ($C2$), and the checking is performed with a given model checker tool ($C3$). The environment definition determines whether the verification is performed completely or just partially. As an answer to our research question $Q1$, there are two main possible outcomes of the verification: either the verification is complete (in which case the pattern is instantiated with the choice of $Strategy$ 2); or, in the case when the system under verification is too complex, compromises are made (i.e. environment restrictions and limited loop unwindings) and thereby the verification is incomplete (in which case $Strategy$ 3 is applied).

Strategy 2: Argument by complete verification using BMC. There are many cases in which verification results obtained with bounded model checking are complete. In these cases the functional correctness of the implementation of property P by SwC is guaranteed by the model checker itself. The bounded model checking verification is complete when the environment constraining the inputs of the model checker Env^{BMC} (verification harness) is relaxed enough to cover all inputs of the environment Env^{SwC} specified by the requirement $ReqId$: $Env^{SwC} \subseteq Env^{BMC}$. In this case, the verification results can be used with the highest confidence as evidence, under the assumption ($A1$).

Strategy 3: Argument by incomplete BMC verification and testing. Due to the complexity of the system under verification, often exhaustive verification is not possible, and the verification is performed under several restrictions of Env^{SwC}, namely Env_i^{BMC}, where $\bigcup Env_i^{BMC} = Env^{BMC}$. There are two orthogonal dimensions in which the environment is restricted: 1) Env^{BMC} restricts the set of possible values taken by the inputs of the software component, or, 2) Env^{BMC} restricts the number of steps which are used to verify the component. In both

cases, only a part of the space of behaviors is covered by the model checker. Consequently, there are behaviors possible in the environment of the component which are specified by the requirement (Env^{SwC}), but not captured in the verification environment (Env^{BMC}). Thus, the assurance deficits caused by incomplete verification must be accompanied by additional evidence in a confidence argument. In the following section, we elaborate on the assurance deficits of incomplete bounded model checking verification and how to compensate for this deficits.

3 Confidence in Incomplete Results

Testing is the most common evidence for functional verification required by safety certification standards. Thereby, in order to be accepted as evidence, the results of a formal verification technique must be shown to be more trustworthy than the results of testing required by the standards [13].

In Figure 3, we describe an argument structure pattern for combining incomplete bounded model checking verification with manually written tests. *Intuitively, the main confidence argument is that the simplifying assumptions under which the bounded model checking is performed are permissive enough to cover test vector sets which satisfy the requirements of the certification standard. If this is not the case, additional test-cases are added to cover the deficits of the bounded model checking verification results (Q2).*

Strategy 3 deals with incomplete bounded model checking verification ($G3.1$) and additional manually written test cases ($G3.2$). The amount of additional

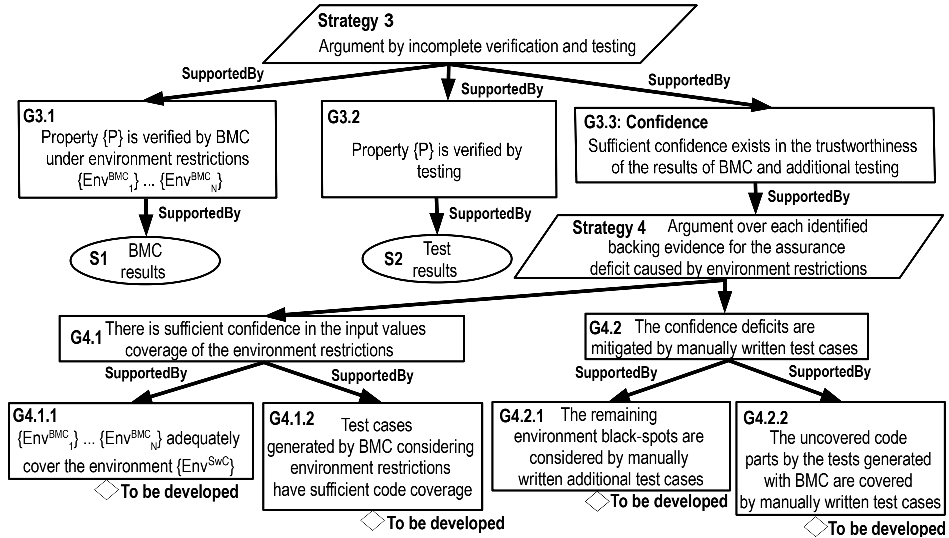

Fig. 3. Pattern for combining the incomplete bounded model checking results with testing. We argue the confidence in bounded model checking results by comparison to testing.

testing should be enough to reach the required confidence ($G3.3$). The strategy for arguing confidence ($Strategy$ 4) is to explicitly mitigate the assurance deficits caused by incomplete bounded model checking due to environment restrictions ($G4.1$, $G4.2$).

Adequacy of environment partitioning. We argue that the sum of environment restrictions ($Env_1^{BMC}, ..., Env_i^{BMC}$) adequately covers the environment Env^{SwC} ($G4.1.1$) – e.g., adequacy can be defined by IEC 61508, which recommends the partitioning of the valid input domain in equivalence classes and the consideration of boundary values. Environment black-spots ($Env^{SwC} \setminus Env^{BMC}$) are parts of the environment which were not covered by the environment definition for the model-checker. These black-spots must be identified, made explicit in an assurance case, and a mitigation method for the risk that they could lead to bugs must be developed. The black-spots are considered by manually writing additional test cases ($G4.2.1$).

Sufficiency of the code covered by bounded model checking Similarly to measuring the code coverage of tests, we can measure the code covered by the bounded model checker when it is run under the verification assumptions. To do this, we use test case generation from the same environment as the verification. Test vectors which satisfy required coverage criteria (e.g. statement, condition, MC/DC) can be generated by the model checker starting from the environment definition ($G4.1.2$). When the required coverage cannot be achieved, it is an indication that the environment restrictions are too narrow. In this case, either the environment must be relaxed such that the required coverage can be reached, or additional test cases must be manually written ($G4.2.2$).

4 Preliminary Experience

In order to operationalize our approach we use the *CBMC* model checker [9] integrated in the *mbeddr* development environment [18]. Besides checking assertions, CBMC also possesses the needed capabilities to generate test cases with a specified coverage. We use the same environment restrictions and CBMC settings to perform the functional verification and to generate test cases. We use mbeddr because it features a user friendly integration of CBMC.

In the following, we present our experience with the verification of three software components, which implement critical functionality. The purpose of our experiments is to investigate the extent to which bounded model checking verification can achieve better coverage than classical testing on software components. These experiments mirror the verification strategies proposed in the patterns.

4.1 Traffic Collision Avoidance System

In our first experiment we verified a software component which implements part of the Traffic Alert and Collision Avoidance System (TCAS) available from the

benchmark algorithms for testing [12]. The TCAS component implements a highly critical functionality because its malfunctioning could lead to collision of planes. The component uses as inputs the positions and speeds of the planes and does not have internal state.

We have checked two properties of the system, namely *P1: Safe advisory selection* and *P2: Best advisory selection*, as in [12]. We have chosen to restrict the values of the variables representing the tracked altitudes of the two planes, based on the constraints on the valid inputs given by the TCAS standard [1]. CBMC managed to fully verify the specified properties under no additional input restrictions in a few seconds and hence obtain 100 percent input coverage. This experiment confirmed us the fact that, with bounded model checking, one can provide, for certain cases, results of exhaustive verification much easier than with any testing method.

4.2 Hamming Error Detection and Correction Algorithm

For our second experiment we chose to verify a commonly used algorithm for detecting and correcting errors based on Hamming codes. We took an algorithm which uses Hamming codes which is based on [16]. This algorithm is representative for a class of error detection and correction algorithms which are often used as parts of the critical functions. Standards like IEC 61508 or ISO 26262 explicitly recommend the use of these algorithms for detecting data failures.

Figure 4 shows an example of a harness definition for the Hamming coding algorithm using mbeddr. On the upper-left-hand side there is an initial definition

```
harness {
  //  Generate random message
  for (uint16 i = 1;i <= size_of_info; i++ ) {
    nondet assign info[i]; constraints {
      info[i] in [0..1]
    }
  } for
  encode_message();
  // inject error
  nondet assign error_pos; constraints {
    error_pos in [1..size_of_transmitted_data]
  }
  transmitted_data[error_pos] ^= 1;
}
correct_transmitted_message();
// check
for (uint16 i = 1;i <= size_of_info; i++ ) {
  assert(info[i] == received_info[i]);
} for
```

```
// Correct error if needed
if (syn != 0) {
  transmitted_data[syn] ^= 1;
} if
```

```
harness {
  //  Generate random message
  for (uint16 i = 1;i <= size_of_info; i++ ) {
    nondet assign info[i]; constraints {
      info[i] in [0..1]
    }
  } for
  encode_message();
  // choose whether to inject the error or not
  nondeterministic_choice: {
    choice: {
      // inject error
      nondet assign error_pos; constraints {
        error_pos in [1..size_of_transmitted_data]
      }
      transmitted_data[error_pos] ^= 1;
    }
    choice: {
      // no error
    }
  }
}
correct_transmitted_message();
// check
for (uint16 i = 1;i <= size_of_info; i++ ) {
  assert(info[i] == received_info[i]);
} for
```

Fig. 4. Environment definition, error injection and verification condition for the Hamming coding algorithm. On the left-hand an initial definition which prevented us to reach branch coverage (left-bottom). On the right-hand side is the corrected environment definition which considers also messages without error.

of the environment – at first we initialize the message to be sent (stored in binary form in the vector `info`), then we encode this message using the Hamming algorithm, we choose an arbitrary position where the error is injected (`error_pos`) and correct the message. The verification condition checks that the initial message is the same as the message decoded upon receival. This harness covers exhaustively all possible vectors of `size_of_info` and all possible one bit errors which can occur within the transmitted vector (`transmitted_data` contains the information together with the corresponding parity bits).

When trying to generate test-cases with branch coverage based on this environment, CBMC could not cover all branches – e.g. the branch from Figure 4-left-bottom was always taken. Manual investigation revealed the fact that our harness did not consider the case when no error happens during transmission. At this point we could have either relaxed the verification environment or we could have manually written some test cases to also verify the uncovered branch. We chose to enhance the harness (Figure 4-right) with a non-deterministic choice to inject/not-inject the error and thereby we could obtain a higher branch coverage. In Figure 5, we present the running time required by CBMC for different lengths of the message. We could exhaustively verify the correct functioning of the algorithm for messages with a length up to 64 bits. Exhaustive testing of these messages would require 2^{58} test-cases and thereby is completely unfeasible. Our conclusion is that bounded model checking can be used to exhaustively check the correctness of the algorithm for relatively small input messages. Correct encoding of messages with a higher length could not be verified by the model checker because the time step bound k was less than the diameter of the transition system that abstractly models the program. In order to cover this assurance deficit, the embedded engineers must manually write additional test-cases, which comply with standards. This experiment shows that the bounded model checker can exhaustively verify cases when the length of the input message is small enough.

number_of_parity_bits	4	6	7
size_of_info	11 bits	58 bits	121 bits
analysis time	2s	60s	> 600s (timeout)

Fig. 5. Time required by CBMC when choosing different lengths of the message to be encoded. CBMC is fast up to messages with total length 64 bits (58 info + 6 parity).

4.3 Case Study 3: Patients Trolley

Our third experiment is the verification of a controller for a smart trolley which assists healthcare professionals in drug administration and other bedside procedures. The smart trolley has several drawers and can serve multiple patients. The trolley responds with different actions to the inputs given by a doctor. We chose this system because 1) it is built as a state machine and can be run in-

finitely 2) it is a safety-critical system because, if it does not function properly, the patient might get the wrong medicine, a fact which might endanger his life.

We chose two properties of the system to verify, namely: *P1: There are never two drawers open at the same time* and *P2: Only the drawers corresponding to the selected patient can be opened.* Both properties come directly from the requirements specification document of the system. Figure 6-left shows the harness definition for the property *P1* – in the main loop we send an arbitrary event to the state machine and we check that in between two events for opening drawers (`EVENT_OPEN_DRAWER`) there is always an event for closing the drawer (`EVENT_DISPLAY_CLOSED`). This only works under the assumption that there is no transition that opens two different drawers at once. This assumption could be checked by code reviewing. In Figure 6-right we present the harness definition for *P2*. We check that the system opens only the valid drawers for a patient.

The state-machine can run infinitely, but we chose to restrict the number of steps in the main loop and thereby the number of events that we send to the state-machine. Thus we performed complete verification up to `MAX_EVENT_NUMBER`. Even with a small value of `MAX_EVENT_NUMBER` we were able to cover all statements of the state-machine. However, the trolley can run for much longer time, and thereby our verification is incomplete. The assurance deficit occurs for long runs of the trolley system. For these cases the developers must use additional manual tests, which comply with standards. The patients trolley example shows the usefulness of bounded model checker based verification on a reactive system which runs infinitely.

```
harness {
  boolean drawer_already_opened = false;
  for (i ++ in [0..MAX_EVENT_NUMBER[) {
    nondet assign crt_event_id; constraints {
      valid_enum(crt_event_id)
    }
    // call the system
    executeEvent(crt_event_id);
    // property: only one drawer can be open
    assert(!(drawer_already_opened &&
        lastEvent == EVENT_OPEN_DRAWER));

    if (state == STATE_DRAWER_SELECTED) {
      if (lastEvent == EVENT_OPEN_DRAWER) {
        drawer_already_opened = true;
      } else if (lastEvent == EVENT_DISPLAY_CLOSED) {
        drawer_already_opened = false;
      }
    } if
  } for
}
```

```
harness {
  for (i ++ in [0..MAX_EVENT_NUMBER[) {
    nondet assign crt_event_id; constraints {
      valid_enum(crt_event_id)
    }
    if (crt_event_id == EVENT_PATIENT_SELECTED) {
      nondet assign patient_in; constraints {
        patient_in in [1..3]
      }
    } if
    if (crt_event_id == EVENT_DRAWER_SELECTED) {
      nondet assign drawer_in; constraints {
        drawer_in in [1..9]
      }
    } if
    // call the system
    executeEvent(crt_event_id);
    // property: only the drawers corresponding to
       the current patient can be open
    if (state == STATE_DRAWER_SELECTED) {
      assert(patient == 1 -> (drawer in [1..3]));
      assert(patient == 2 -> (drawer in [4..6]));
      assert(patient == 3 -> (drawer in [7..9]));
    } if
  } for
}
```

Fig. 6. Environment definition and verification condition for the Smart Trolley system: on the left we check property *P1* and on the right we check property *P2*.

5 Discussion

On simplifying assumptions. In practice, there are a multitude of factors that must be considered when analyses tools replace the execution of the tests on the target hardware. For example, the C compiler used to produce the binary from the sources might have itself bugs or have a different interpretation of corner cases of the C language than the verification tool. Furthermore, hardware particularities like the endianess, word length or memory model must be treated soundly by the verification tool. These aspects are not in the scope of this paper, but should be thoroughly considered during the development of a real-world assurance case.

On automating the complementary testing. When the bounded model checking is incomplete, we propose to cover the assurance deficits using manually written test-cases. However, recent developments in verification based on conditional model checking [7] are able to characterize the state space of the program covered by the verification tool and use this information to generate test-cases for the uncovered parts [10]. The information about the uncovered code parts can be used in the confidence argument and part of the test vectors can be obtained in an automatic manner.

On using tests vectors generation to measure confidence. One of the means we proposed to measure the completeness degree of incomplete verification is to generate test vectors starting from the same enviromental assumptions. However, empirical studies on the effectiveness of coverage-directed tests generation to uncover bugs show disappointing results [17]. Thereby, structural coverage criteria can indicate weaknesses in our assumptions (when these criteria are NOT fulfilled) and offer only a weak confidence in the verification when the criteria are fulfilled.

On practicality and costs. Our approach builds a bridge between two extreme cases. The first case is when the model checker can explore the space of behaviors exhaustively; the second case is when the model checker cannot produce any meaningful result, even when a narrow environment is used. In the first case, the verification is complete; in the second case, we rely completely on the results of traditional testing. In this paper, we argue for a middle way to complement the verification results with testing. Finding a sweet-spot, in which the cost-benefits of applying formal verification is the highest, is of a paramount importance for the adoption of the approach, especially for functions at lower criticality levels.

On using the CBMC bounded model checker. CMBC is still in need of verification and validation in order to be used as assurance evidence generator. However, the reason for using CBMC in our work is that it provides out-of-the-box features which are key enablers for our approach. Firstly, the CBMC analyses are bit-precise and thereby accurate. Secondly, CBMC offers the possibility to instrument loops (and recursion) to detect insufficient unwindings and to warn

about incomplete results. Thirdly, CBMC offers the function to generate tests with a given code coverage and we use these tests as backing evidence for the coverage degree of the bounded model checking verification. Last but not least, CBMC allow the specification of verification environment. There are, however, other bounded model checking tools which could be used instead of CBMC.

6 Related Work

Formal verification for assurance. Habli [13] and Denney [11] present a generic safety argument for the use of formal methods results for certification. Basir [5] proposes a method to derive safety cases from formally verified code using Hoare-style inferences. Bennion [6] develops an assurance case for arguing the compliance of the Simulink Design Verifier model checker to DO-178C. We propose an argument structure pattern for using successful, but possibly incomplete bounded model checking results as certification evidence. For the cases when the verification is incomplete, our pattern uses results of additional testing as evidence and comprises a confidence argument.

Confidence of evidence. Habli [13] emphasizes the need of including all the known limitations of the used formal verification technique in order to achieve trust in the results. Hawkins [14] defines assurance deficits as a prohibiting factor of perfect confidence in a claim about an assurance evidence. Ayoub [4] proposes the usage of separate argumentation legs for arguing that certain confidence exists in a certain assurance evidence. This is done by explicitly listing identified assurance deficits and the measures taken against them. They call these argumentation legs confidence arguments. The usage of complementary diverse evidence is encouraged by Littlewood [15], who demonstrates an increase of confidence in the argument about the system safety when having both a verification and a testing argument leg. We propose a confidence argumentation structure for explicitly describing the assurance deficits of this verification method and for providing corresponding backing arguments.

Complementing verification with testing. Conditional model checking [7] is a technique to characterize the state space of the program which was covered by the model checker and use this information for subsequent analyses or to generate test cases for the uncovered parts Czech[10]. Christakis [8] uses a similar technique in order to explicitly specify all assumptions which the verification engine performed and thereby, to enable collaborative verification. The focus of these works is on making the deficits of model checking explicit and cover these deficits by other verification methods. The above mentioned works are complementary to our work and they can be used to better characterize the confidence in incomplete results, to increase the automation of the tests generation, or to use other complementary verification methods to minimize the deficits.

7 Conclusions

In this paper, we presented an approach to use successful results of software bounded model checking in an assurance case. We propose to use additional testing to mitigate the possible assurance deficits of incomplete bounded model checking. Our longer term goal is to enable practitioners who develop safety critical systems to benefit from the bounded model checking technology. As future work, we plan to investigate in detail heterogeneous backing evidence from other verification methods (e.g., code review) to reinforce incomplete model checking results.

Acknowledgments. The research leading to these results has received funding from the European Union's Seventh Framework Programme FP7/2007–2013 under grant agreement n°610640.

References

1. *Introduction to TCAS II version 7.1*, November 2000.
2. International standard IEC 61508, 2008.
3. GSN community standard version 1. Technical report, Nov. 2011.
4. A. Ayoub, B. Kim, I. Lee, and O. Sokolsky. A systematic approach to justifying sufficient confidence in software safety arguments. In *SafeComp*, 2012.
5. N. Basir, E. Denney, and B. Fischer. Constructing a safety case for automatically generated code from formal program verification information. In *NFM*, 2010.
6. M. Bennion and I. Habli. A candid industrial evaluation of formal software verification using model checking. In *ICSE*, 2014.
7. D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler. Conditional model checking: A technique to pass information between verifiers. In *FASE*, 2012.
8. M. Christakis, P. Müller, and V. Wüstholz. Collaborative verification and testing with explicit assumptions. In *FM*, 2012.
9. E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, 2004.
10. M. Czech, M. C. Jakobs, and H. Wehrheim. Just test what you cannot verify! In A. Egyed and I. Schaefer, editors, *FASE*. Springer, 2015.
11. E. Denney and G. Pai. Evidence arguments for using formal methods in software certification. In *WoSoCer*, 2013.
12. A. Gotlieb. TCAS software verification using constraint programming. *Knowledge Eng. Review*, 2012.
13. I. Habli and T. Kelly. A generic goal-based certification argument for the justification of formal analysis. *Electron. Notes Theor. Comput. Sci.*, 2009.
14. R. Hawkins, T. Kelly, J. Knight, and P. Graydon. A new approach to creating clear safety arguments. In *Advances in Systems Safety*, 2011.
15. B. Littlewood and D. Wright. The use of multilegged arguments to increase confidence in safety claims for software-based systems: A study based on a BBN analysis of an idealized example. *Software Engineering, IEEE Transactions on*, 2007.
16. R. H. Morelos-Zaragoza. *The Art of Error Correcting Coding.* John Wiley & Sons.
17. M. Staats, G. Gay, M. Whalen, and M. Heimdahl. On the danger of coverage directed test case generation. In *FASE*, 2012.
18. M. Voelter, D. Ratiu, B. Kolb, and B. Schätz. mbeddr: instantiating a language workbench in the embedded software domain. *Autom. Softw. Eng.*, 2013.