Arguing on Software-level Verification Techniques Appropriateness

Carmen Cârlan, Barbara Gallina, Severin Kacianka, and Ruth Breu

fortiss GmbH, Munich, Germany email: carlan@fortiss.org
Mälardalen University, Västeras, Sweden email: barbara.gallina@mdh.se
Technische Universität München, Garching, Germany email: kacianka@in.tum.de
Institut für Informatik, Innsbruck, Austria email: ruth.breu@uibk.ac.at

Abstract. In this paper, we investigate the pondered selection of innovative software verification technology in the safety-critical domain and its implications. Verification tools perform analyses, testing or simulation activities. The compliance of the techniques implemented by these tools to fulfill standard-mandated objectives (i.e., to be means of compliance in the context of DO-178C and related supplements) should be explained to the certification body. It is thereby difficult for practitioners to use novel techniques, without a systematic method for arguing their appropriateness. Thus, we offer a method for arguing the appropriate application of a certain verification technique (potentially in combination with other techniques) to produce the evidence needed to satisfy certification objectives regarding fault detection and mitigation in a realistic avionics application via safety cases. We use this method for the choice of an appropriate compiler to support the development of a drone.

Keywords: safety cases, faults, standard compliance, verification techniques

1 Introduction

For the certification of safety-critical systems, safety engineers are frequently required to present a safety case of the system. A safety case is *a documented body of evidence that provides a convincing and valid argument that a system is adequately safe for a given application in a given environment* [3]. The certification authority investigates the confidence in the claims of a safety case, namely the probability of the claim being true [3]. This probability depends on how uncertainties regarding the safety case (e.g., regarding the evidence expected to support the claims) are handled. For example, one uncertainty regards the correctness of the implementation. This uncertainty lays in the verification procedure and is caused by 1) uncertainty in the correct implementation of the verification tool – *Can the output of the tool be trusted?*, 2) uncertainty in the rationale of the verification technique – *Is this the right way for verifying the fulfillment of system requirements?* [7]. Thus, in order to employ a state-of-the-art verification tool, the engineer needs to assess the *appropriateness* of the technique it implements. A technique is *appropriate* if it provides *trustworthy* and

relevant verification results. Considerable research effort has been put into the investigation of arguing tool assurance (i.e., integrity and qualification according to standard) [7]. Safety standards provide certain objectives techniques must satisfy. These objectives are typically used in industry in form of checklists for the selection of verification techniques. However, the standards do not clarify 1) why the objectives contribute to demonstrate the confidence in results and 2) how they relate to the characteristics of the verification technique that must achieve them. For example, before DO-333 [22], it was unknown 1) what was the relevance of this objective for the system's safety, and 2) how the testing structural coverage objective could be addressed with a formal verification technique. This problem has been dealt with by supplements providing guidance on how to adapt these innovative technologies to a DO-178C project (e.g., DO-333 Formal Methods Supplement). However, the creation of such supplements can take years. One of the main causes for this is that little is known about arguing whether a technique is appropriate to support a given activity. The appropriateness of a technique is its quality to satisfy the corresponding objectives. Thus, in this paper, we take on the problem of *appropriately employing verification techniques for the construction of systems according to a specific level of stringency, specified via an assurance level (AL).*

This paper's contribution at tackling this problem is three-fold. First, we offer an alternative for the pondered selection of verification techniques. *Pondered selection* means a selection of techniques, based on how they are contributing to typical systematic failure avoidance. We achieve this by extending the Structured Assurance Case Metamodel (SACM) [19]. Second, we describe relationship types between heterogeneous verification results collaborating to the achievement of one safety goal. Third, based on this meta-model, we provide safety case patterns for arguing the appropriateness of a certain technique.

In Section 2, we provide the context of our problem statement. In Section 3, we present our metamodel. Then, in Section 4, we present a set of safety case patterns that help with the pondered selection of a certain verification technique for the performance of a certain activity. We evaluate our approach by instantiating the proposed metamodel to assess the appropriateness of the results from a compiler, performed on a drone (see Section 5). The last two sections contain related work and conclusions.

2 Background

The development of safety-critical systems is guided by standards. In avionics, it is recommended that software developers reach the objectives defined in the **DO-178C** de-facto standard [21]. DO-178C is technology-independent, abstractly defining development and verification activities to be performed. For each activity, it defines a set of objectives that need to be satisfied by concrete verification techniques. How these objectives are to be fulfilled is up to concrete means of compliance. A means of compliance is the technique that the developer uses to satisfy the objectives stated in the standard [21]. The certification

authority needs to agree on the means of compliance proposed by the developer. Techniques of the software verification process need to be proposed during the certification liaison process. The verification process has two purposes: 1) show that the system implements its safety requirements and 2) detect and report faults that may have been introduced during the software development processes. DO-178C focuses on analysis, reviews and software testing techniques for performing verification activities. Whereas its supplement DO-333 [22] provides guidance for using formal methods (e.g., abstract interpretation, model checking, theorem-proving and satisfiability solving) in the certification of airborne systems. The supplement modifies DO-178C objectives, activities, and software life cycle data to address when formal methods are used as part of the software development process.

The Object Management Group offers a standardized modeling language for describing safety cases: the **Structured Assurance Case Metamodel (SACM)** 2.0 [19]. SACM contains a structured way of describing evidence-related efforts, namely the Artefact Metamodel. The Artefact Metamodel contains classes depicting the following: artefacts, participants, resources, activities, and techniques. The *Activity* class represents units of work related to the management of *ArtefactAssets*. The *Technique* class describes the techniques performing the activities. For example, the *Verification of Low-level Requirements* activity, from DO-178C/DO-333, may be performed either via a testing technique (e.g., unit testing) or a formal analysis technique (e.g., theorem proving). There are three types of evidence in safety cases, namely *direct*, *backing* and *reinforcement* [24]. Whereas direct evidence refers to proofs that the system under certification meets the safety goal, the backing evidence proves that the direct evidence can be used in the argumentation with confidence [16, 1]. In order to justify the confidence, the relevance of the evidence serves as a proof of safety claim's satisfaction [24]. The relevance of an evidence type is assessed by documenting the *role* and *limitations* of the underlying technique. Such limitations may be that the verification technique is not able to cover the entire input space or to identify deep faults due to insufficient unrolling. This information helps to make an informed decision when choosing a type of evidence for satisfying a certain claim [16].

3 Pondering the Selection of Verification Techniques

In this section, we propose guidelines for the selection of techniques in order to fulfill verification objectives of DO-178C/DO-333. We propose a metamodel that offers a description of standard-mandated compliance and considers the relationships between standard objectives, software verification techniques and safety evidence. Our metamodel extends the SACM Artefact Metamodel. We refine the *Technique* and *Activity* classes in the SACM metamodel, by *Verification Technique* and, respectively, *Verification Activity* classes (see Fig. 1).

Next, we will explain the attributes pallet of our proposed SACM *Verification Technique* class, depicted in Fig. 1. These attributes enable the characterization

of the appropriateness of a verification technique. Some of the attributes come from the objectives a technique needs to fulfill in order to be employed for a DO-178C/DO-333-compliant activity (e.g., structural coverage). Additional attributes are taken from specialized literature depicting safety verification techniques (e.g., works such as [2], [23]). These attributes are taken into consideration during safety assurance, but have not been documented in the standard and are typically not included in checklists, since they are considered implicit attributes (e.g., technique soundness). Explicitly addressing these attributes helps at taking an informed decision and to build a convincing argument for the technique appropriateness. For some of the described attributes, we provide enumerations of the values the attributes can take. The permitted values are extracted from the standard.

Any verification technique has a certain name, which uniquely identifies it (see *techniqueName* attribute). Verification techniques address different verification objectives because of their different rationale. Thus, the *Verification Technique* class should contain an attribute referencing a description of the rationale (see the *verificationRationale* attribute). Documenting the context is required when assessing how appropriate the technique is and indicates the feasibility of the technique's application. For instance, the *exampleApplication* attribute, referencing projects which already made use of the technique, can only be used if the verification's context is similar. The context includes the constraints on the environment of the system (see *envConstraints*) [4]. This attribute ensures that

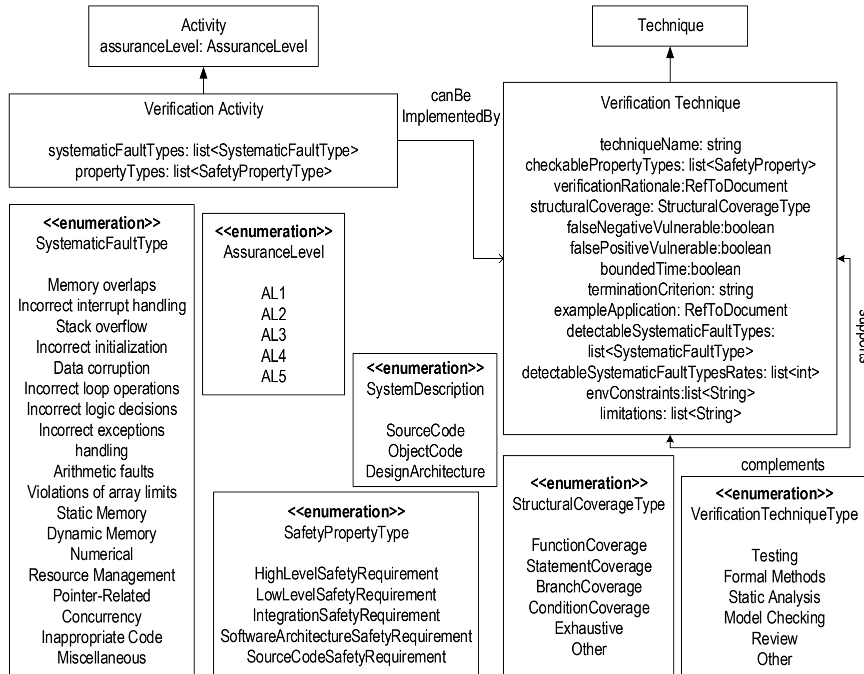

Fig. 1. A meta-model for depicting verification techniques and activities

the technique is appropriate for verifying the system, in the context in which the system is supposed to work.

No unique technique can cover all objectives of the verification process. For example, the rationale of **testing** techniques is to execute the code to reveal faults. However, testing has its limitations. One limitation is that testing techniques may detect concurrency faults, but overall, the level of confidence about the quality of the code is not high. This is due to the fact that concurrent software is inherently non-deterministic. Thus, it is necessary to describe the objectives of the verification technique, namely the types of properties it can check (see *checkablePropertyTypes* attribute). The *checkablePropertyTypes* attribute enumerates all the standard-mandated properties the respective technique can cover.

Static analysis takes all thread execution pathways and deployment scenarios into account. Thus, **static analysis** verification techniques are more appropriate for discovering and diagnosing concurrency faults. Hence, it is important to know to what extent the scope may be verified by the technique (i.e., number of explored states or loop unrolling). Thus, we add to the *VerificationTechnique* class the *structuralCoverage* attribute. Static analysis techniques have the benefit of verifying the system's code exhaustively. However, they also have their limitations, which need to be specified, understood and assessed for the fulfillment of an objective (see *limitations* attribute). In the *Verification Technique* class we suggest possible limitations of verification techniques. One possible limitation is the detection of false negatives (see attribute *falseNegativeVulnerable*). This does not affect the certification process per se, but it delays the process, since the safety engineer might investigate an error which does not exist. Another limitation may be that a verification technique verifies a property and may say that the property is satisfied, when it is not. Such techniques are unsound. We capture this limitation in the *falsePositiveVulnerable* boolean attribute. Example of false-positives is when a test result wrongly indicates that a particular condition or attribute is present. Unsound techniques cannot be used in the verification process of safety-critical systems, since they may be harmful. For example, the consequences of falsely confirming that a state is reachable may be catastrophic. Thus, verification techniques that have the *falsePositiveVulnerable* attribute set to "true" may not be used for employing safety verification activities. Furthermore, a frequent type of limitation, which is specific to model checking techniques, is the fact that the verification might not terminate in due time. In such cases, model checking cannot be set to explore all the states due to their large number: large number of states makes the verification last longer than feasible (see the boolean *boundedTime* attribute). If the *boundedTime* attribute is true, the *terminationCriterion* attribute should also be set, in order to know when the verification is supposed to stop and to know up to which length counterexamples have been searched.

The verification process is described by DO-178C as a process for discovering faults. Thus, in order to analyze a verification technique, it is important to know what types of faults it may uncover - *detectableSystematicFaultTypes* (e.g., arithmetic faults, violations of array limits). In studies presenting tools checked

against benchmarks, there is a so called defect rate, which refers to the percentage of erroneous tests. The *detectableSystematicFaultTypesRates* attribute of the *Verification Technique* class offers a quantitative assessment of how a selected technique is better than others. Each element from the *detectableSystematicFaultTypesRates* list corresponds to the element with the same position from the *detectableSystematicFaultTypes* list.

This metamodel describes a set of traces between two different aspects of safety argumentation artefacts: the verification activity and the verification technique (see Fig. 1). These traces enable the assessment of the appropriateness of a technique for performing some activity. The *Verification Activity* class, depicted in Fig. 1, offers a general structure for depicting any verification activities in DO-178C. The attributes of this class are derived from the structure of these verification activities. As presented in [11], every development activity, including verification activities, has an assurance level. In the context of DO-333 it is called assurance level (AL). Thus, we add to the *Activity* class the *assuranceLevel* attribute. The *Activity* class also depicts the typical faults that are to be identified and mitigated during the referred activity type (see the attribute *systematicFaults*). The *propertyTypes*, which must be achieved by the verification process, given the assurance level, are indicated by *Tables A 3-6*, in DO-178C. A verification technique is appropriate to perform a certain verification activity (see *canBeImplementedBy* relationship), if the technique is able to check *at least some* of the *propertyTypes* required by the activity to be checked (i.e., the set *checkablePropertyTypes* of the technique is at least a subset of the *propertyTypes* set of the activity), applicable by the *assuranceLevel*. Also, the technique should be able to detect *at least some* of the *systematicFaultTypes* specified by the activity (i.e., the set *detectableSystematicFaultTypes* of the technique is at least a subset of the *systematicFaultTypes* set of the activity). However, a technique may only be able to check *certain* properties or to identify *certain* faults. Different techniques may be combined in order to perform a verification activity. For example, in Cârlan et al. [6], we present a testing technique and a model checker collaborating for discharging verification goals.

Heterogeneous Verification Techniques. Evidence tends to be incomplete (e.g., a single test case, or model checking of a single property). In this situation, multiple items of evidence are needed. In Fig. 1, we document and reason about the relationships between the heterogeneous verification techniques generating evidence items. One type of relationships is *supports*. This relationship covers the case where a verification technique is used to assess the fulfillment of an objective by the results of another verification technique. Techniques in a *supports* relationship work orthogonally, addressing different concerns – one discharges safety goals, the other is used to verify the results discharging the safety goals, in order to assess the trustworthiness of the evidence provided by the first technique. Thus, a verification technique *supports* another if it is used to detect faults in the other's verification results, by providing *backing* evidence. For example, a model checking technique may support a static analysis technique by verifying the faults detected [5]. The other relationship type is *complements*.

This relation represents two verification techniques that collaborate for providing relevant evidence for discharging together safety goals. On the one hand, a verification technique may complement another if it is used to detect the faults not identified by the other. On the other hand, a verification technique complements another technique if it is able to verify types of requirements which cannot be verified by the other technique. Both of the techniques provide *direct* evidence. For example, verifying a set of properties via bounded model checking, combined with testing [6].

4 A Pattern for Arguing Technique Appropriateness

In the system's safety cases, the developer has to argue that the verification results are *trustworthy* and *relevant*. As mentioned in Sec. 1, this enables the assessor to have confidence in the results. The techniques associated with the creation, inspection, review or analysis of assurance artefacts contribute to the level of relevance of the safety case evidence [19, 8]. In this section, we offer a pattern for arguing the appropriateness of verification techniques, driven by the need to deliver relevant safety evidence. We call this argumentation structure the *technique appropriateness* argument pattern. Each element of the pattern relates to an attribute of our proposed metamodel. The attributes from the metamodel are italicized in the safety claims. The fact that the pattern is based on the metamodel eases the (semi-)automatic pattern instantiation.

The top-level goal of the pattern depicted in Fig. 2 is that the technique implemented by the tool employed in the execution/automation of a certain activity is appropriate for generating activity outputs (*G1*). Goal *G1* may only be satisfied if the technique is sound (*C4*). In order to argue over the capabilities of a technique for discharging safety goals, one should explicitly state the verification scope and the environmental constraints (*C1*, *C2*). The fact that the verification technique has been previously used in other projects with similar environmental constraints may be used as justification for its appropriateness. Each verification technique needs to demonstrate the satisfaction of several goals (i.e., required objectives to be fulfilled and outputs to be provided), as defined in *Tables A 3-7*, from DO-178C (*C3*). The main goal of the performed verification activity is the verification of a certain type of requirements (*G3*), as recommended by the *assuranceLevel* (*C7*). One should argue this using the rationale of the technique (*G6.1*). The argumentation further developing goal *G6.1* mirrors the verification steps (presented in [5]). When arguing over goal *G3*, it is relevant to cover the requirement types that are imposed by the activity type to be detected by the technique (*C6*). Each requirement type has different suitable verification techniques. Thus, the selected verification technique may not be able to cover all the requirement types (*G3.1*). When the technique cannot cover all the verification space (i.e., to have 100 percent structural coverage), another technique may be employed to cover the rest of the verification space, as stated in goal *G6.2*. However, this is an optional goal, since structural code coverage is not an applicable coverage criterion for all verification techniques (e.g., de-

ductive verification). When arguing over goal *G1*, one should also consider the limitations (weaknesses) implied when employing the respective technique (*S4*). Techniques may work together to compensate for such limitations (*G4.1*, *G4.2*, *G4.3*). *Table A-7* from DO-178C and DO-333 recommends that any verification results should be verified (*G1.1*).

A considerable set of standard-mandated compliance requirements for verification techniques targets the detection of certain typical faults (see *G2*, in Fig. 2). Indeed, if a verification technique does not eliminate any fault, the performed verification activity does not increase the confidence in the claim [13]. While arguing for the main goal *G1*, the capability of detecting (some) typical faults is also relevant (*G2.1*). If the selected verification technique cannot detect some of expected fault types, it must be supported by another verification technique (*G2.2*). Whereas all the other sub-goals of *G1* offer a *mere compliance* to standard-mandated objectives, this part of the safety argumentation (*G2.1*) offers *pondered compliance* (i.e., aware selection of techniques). The contribution to failure avoidance is two-fold 1) the coverage of the typical fault types that are imposed by the activity type - qualitative argument (*G5*) and 2) the number of detectable faults - quantitative argument (*J2*). The strength of the technique is given by the number of implementation problems (faults) types it can detect. For arguing the coverage of a typical fault type, an argument based on the *fault-based argumentation* pattern depicted in Fig. 3. The scope of the *fault-based argumentation* pattern is to offer a structure for arguing the selection of a certain technique, by stating its contribution to failure mode avoidance/reduction.

5 Example

In this section, we present our experience with selecting an adequate open-source compiler for a drone in compliance with DO-178C. In the context of safety critical projects, there are few compiler selection approaches [25]. DO-178C compliant software may not contain software faults that lead to failure. Compilers are designed to perform minimal static analysis on the program in order to detect software faults [17]. In a project involving high costs, where *time is money*, engineers should take advantage of the static analysis techniques provided “for free” by compilers. Cârlan et. al. [5] present a code review workflow, which employs a set of static analysis for discharging safety goals. Instead of employing a large number of expensive static analysis, we want to also rely on the used compiler(s) for detecting some of the software faults and thus possibly reducing the size of the static analysis set. According to Höller et. al. [17], diverse compiling is able to detect a larger number of software faults than single compiling. For example, diverse compiling can help to detect up to about 70 percent of memory-related software faults. These indicate that 1) different compilers implement different static analysis techniques and 2) static analysis techniques underlying compilers may play a significant role in the verification process. Different static analysis techniques embedded in different compilers are appropriate to detect different types of faults. Thus, rather then selecting an adequate compiler, we will select

C4: Technique {t} is not false positive vulnerable

C2: Enviroment constraints *envConstraints*

G1: Mere compliance Technique *{techniqueName}* is an appropriate means of compliance for verification activity {a}, in order to achieve *{assuranceLevel}*

C3: DO-178C Tables A 3-7

C1: System element under verification {suv}

J1: Verification technique *{techniqueName}* has been succesfully used in *{exampleApplication}*

S4 Argument over limitations of *{techniqueName}*

C5: *{limitations}*

falseNegativeVulnarable = true

G4.1: Supporting heterogeneous evidence Supporting heterogeneous verification technique {vt6} is used for mitigating the fact that *{techniqueName}* is false negative vulnerable

boundedTime = true

G1.1 Verification of verification technique {vt} results was performed

optional

G1.1.1: Heterogeneous backing evidence Verification tool {vt1} supports *{techniqueName}* by independently verifying ist results

C6: Systematic fault types to be detected: *{systematicFaultTypes}*

S1 Argument over activity-specific objectives

S2 Argument over activity-specific systematic fault types to be detected

S3 Argument over activity-specific property types to be checked

G2: Pondered compliance The *detectableSystematicFaultTypes* that need to be revealed while performing activity {a} are detected

C6: Propery types to be checked: *{property Types}*

G3 Verification technique *{techniqueName}* checks activity-specific property of type {pt}

G4.2: The effects of the limitation have been mitigated

G4.3: The effects of the fact that the technique is bounded have been mitigated

G2.1: Pondered compliance (Part of) the systematic fault types that need to be revealed while performing activity {a} are detected by *{techniqueName}*

G2.2: Complementary heterogeneous evidence for fault coverage Complementary heterogeneous evidence from verification technique {vt2} is used for detecting the fault types that cannot be detected by *{techniqueName}*

S6 Argument over activity-specific properties of type {pt}

G6.1 Argument over verification rationale

C7: Applicable by {assurance*Level*}

structuralCoverage < 100%

S5 Argument over detectable systematic fault types detected by *{techniqueName}* during activity {a}

C8: Detected Systematic fault types: *{detectableSystematic FaultTypes}*

G5: Defect-based argumentation Detectable systematic faults of type {t} have been detected via *{techniqueName}* and mitigated during activity {a}

G3.1: Complementary heterogeneous evidence Complementary heterogeneous evidence from verification technique {vt3} is used for checking activity-specific property of type {pt}, when {pt} is not comprised in *{checkablePropertyTypes}*

G6.2: Complementary heterogeneous evidence Complementary heterogeneous evidence from verification technique {vt5} is used for veryifing the part of the system artefact not comprised in structural coverage of *{techniqueName}*

S1 *{technique Name}* results

Fig. 2. Technique appropriateness argument pattern

a compiler which integrates a static analysis technique appropriate for *complementing* static analysis techniques.

As compilers may merely offer some simple static checks, a static analysis technique implemented by a compiler may only have a *complements* relationship to a verification technique, which is able to performing the verification activity. In order for the compiler to complement a verification technique to perform a particular activity, its underlying static analysis technique should be capable of detecting some of *systematicFaultType*s that need to be detected during the respective activity. For brevity reasons, in this paper, we investigate the support a compiler offers for the performance of one activity, namely the *6.3.4 Reviews and Analyses of Source Code* activity. For the pondered selection of the static analysis technique underlying a compiler, we modeled this activity in accordance to the *Verification Activity* class, presented in Section 3 (see Fig. 4). This activity checks the system at source level. The types of properties that should be checked during this activity (*checkablePropertyTypes* attribute), together with the objectives to be fulfilled by the performance of this activity (see *detectableSystematicFaultTypes*) are taken from the DO-178C standard. The *detectableSystematicFaultTypes* attribute is filled with information given by the *f. Accuracy and consistency* paragraph of the *6.3.4*.

The battle between clang and gcc. We have two candidate techniques for the role of compiler in our project, namely clang and gcc. We model the static analysis techniques implemented by these two compilers based on our proposed metamodel in Section 3. Compilers are able to check the entire code, with no exceptions, hence the *structuralCoverage* for both compilers is depicted as *Exhaustive*. We based our decision also on the experience of RV Team, while compiling the code from the Toyota Benchmark [23] with these two compilers (see the *exampleApplication* attributes). The *detectableSystematicFaultTypes* attribute, together with the *detectableSystematicFaultTypesRates* attribute for both of the models have been filled in with information from the same experience report. While selecting the appropriate static analysis, its impact on the worst-

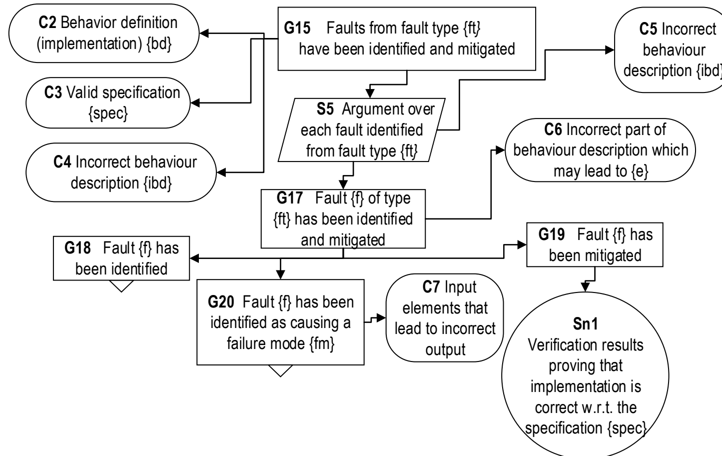

Fig. 3. The fault-based argumentation pattern

case execution timing should be considered and assessed (see *potentialDeficits* attribute). In Fig. 4, we see that both of the techniques are able to detect *systematicFaultTypes*, which should be detected during the *6.3.4* activity. This makes them equally appropriate candidates. However, while the technique implemented by the clang compiler may detect defects of type static memory (e.g., static buffer overrun/underrun), the technique in gcc compiler does not have this ability. In turn, the gcc technique is capable of finding imperfect code defects, such as dead code detection, floating-point arithmetic, use of uninitialized variables, unused variables and improper error handling. The selection is now reduced to selecting the type of defects that would have a bigger impact on the safety of the system under verification (in our example the drone). In our concrete case, static buffer overflows are a bigger concern than floating-point arithmetic defects, because they may lead the vehicle's software to crash. As learned from the Ariane 5 accident, buffer overflows may have devastating consequences on a flying system's safety [12]. As such, we selected the static analysis of clang compiler and, implicitly, the clang compiler. From this experience, we learned that, in the selection of a verification technique, it is not only important what kind of faults a verification technique is able to detect, but also the impact of the type of fault on system safety.

Proving the selection. In order to confirm that we chose the appropriate compiler, we compile a small sample of code in the environment in which we will compile the code for the drone, namely the robot operating system (ROS) [20]. To simplify the discussion, we show the problems on a much smaller ROS introductory example, *turtlesim*[1]. This example allows the user to control an animated turtle by sending it ROS messages. In principle the control software for the UAV[2] uses the same mechanisms and build environment. We mutated that code with four buffer overflow defects (see Fig. 5). We observe that, as our scope

[1] `http://wiki.ros.org/turtlesim`, the source code can be found on github: `https://github.com/ros/ros_tutorials`

[2] We used the Erlecopter: `http://erlerobotics.com/blog/erle-copter/`

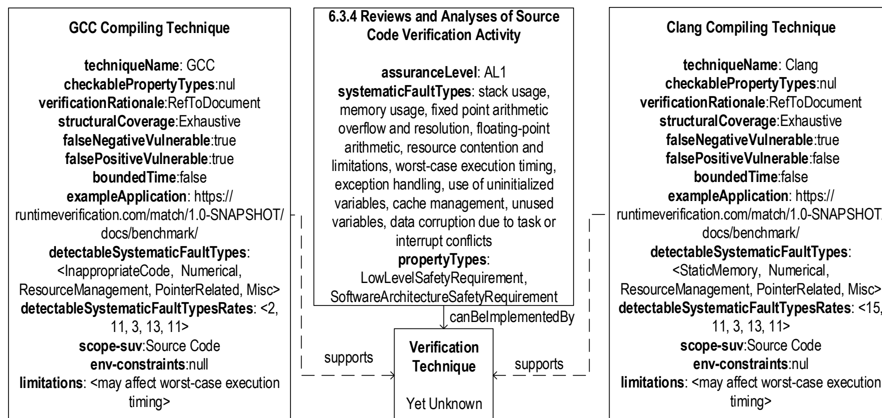

Fig. 4. A model for clang and gcc and the DO-178C 6.3.4 verification activity

```
// mList is declared as float mList[4];
void Turtle::velocityCallback(const geometry_msgs::Twist::ConstPtr& vel)
{ last_command_time_ = ros::WallTime::now();
  lin_vel_ = vel->linear.x;
  ang_vel_ = vel->angular.z;
  // remeber the last 5 velocities to replay them
  mList[0] = lin_vel_; mList[1] = mList[0];
  mList[2] = mList[1]; mList[3] = mList[2];
  // static buffer overflow
  mList[4] = mList[3]; }
```

Fig. 5. Mutated code of method that sets the velocity values for the turtle

was to have a compiler supporting the detection of buffer overflow faults, clang was more appropriate, since it discovered all the four faults (see Fig. 6 for an example), whereas gcc was not able to discover any buffer overflow faults.

```
/home/user/catkin_ws/src/turtlesim/src/turtle.cpp:72:2: warning: array index
     4 is past the end of the array
       (which contains 4 elements) [-Warray-bounds]
         mList[4] = mList[3];
         ^      ~
/home/user/catkin_ws/src/turtlesim/include/turtlesim/turtle.h:79:3: note:
     array 'mList' declared here
  float mList[4];
  ^
1 warning generated.
```

Fig. 6. The error message given by clang. gcc does not point out the error

In Fig. 7, we show how clang compiler contributes at discharging the main goal of the *Technique appropriateness argument pattern*. Goal *G2.2.1* is to be further-developed by instantiating the *Fault-based argumentation pattern*. All the warnings from the compiler are to be documented and referenced in the documentation. We suggest building a test case for every warning. After dealing with these warning, in order to prove that they have been mitigated (see *G19* from the *Fault-based argumentation pattern*), we would run the test cases and reference their results.

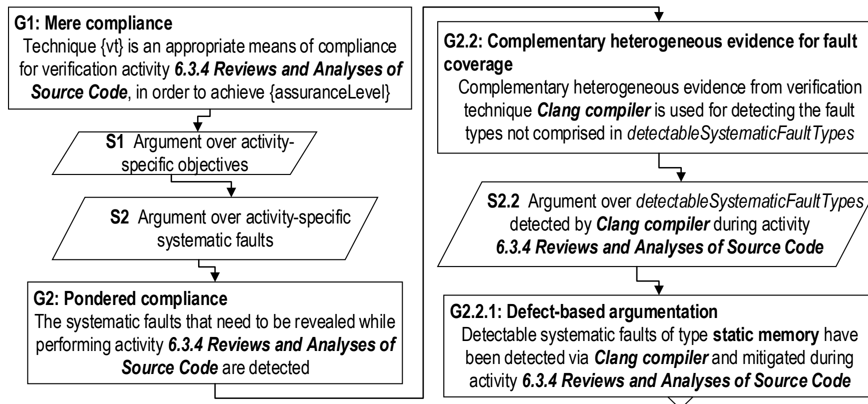

Fig. 7. Partial instantiation of *Technique appropriateness argument pattern*

6 Related Work

The problem of arguing compliance with standards by using patterns has been investigated quite heavily during the last decade. Habli et al. [15] and Denney et al. [7] present safety case patterns for the use of formal methods results for certification. Bennion et al. [2] present a safety case for arguing the compliance of the Simulink Design Verifier model checker to DO-178C. Gallina et al. [9] argue about adequacy of a model-based testing process. Cârlan et al. [5] offer a pattern which integrates static analysis results in an argument for the fulfillment of certain safety objectives. While all these works focus on a certain verification technique as strategy for discharging a safety goal, we offer a safety case pattern to argue the pondered selection of a verification techniques of any type for discharging a safety goal.

Similarly, the problems related to compliance of the certification artefacts and their confidence have also been tackled. Gallina et al. [10] proposes a process compliance pattern for arguing about reuse of tool qualification certification artifacts. One of the identified sub-goals for the claim of trustworthy performance of a certain tasks is that the guidance (how the activity should be performed) has been followed. We offer a reusable argumentation structure for the appropriateness of a technique for a certain activity. We argue that the technique implemented by that tool follows the guidance given by the standard. The problem of confidence in the certification artefacts has been addressed by Graydon et al. [14], who offer a framework for utilizing safety cases for the selection of certain technologies for building safety-critical systems. How to make a decision is, however, not explained. We propose one criterion for making justified decisions on used verification technologies, namely that they need to contribute to the identification or the mitigation of systematic faults known to affect systems' safety. Holloway [18] presents safety case patterns mirroring DO-178C software correctness objectives. In contrast to the work of Holloway, we present safety case patterns, which are built mirroring the important characteristics to be compliant with the DO-178C verification objectives.

7 Conclusions

The output of a tool implementing a certain verification technique may be used as evidence in a safety case. For this, one needs to assess if the respective verification technique is appropriate to generate results for supporting the truth of safety case claims. In this paper, we proposed a metamodel to provide guidelines for the pondered selection of appropriate verification technologies for performing standard-mandated verification activities. Based on this metamodel, we also presented a set of safety case patterns arguing the appropriateness of the verification techniques providing assurance evidence. As future work, we plan to validate our proposed pattern by applying it to argue about appropriateness of verification techniques used in the projects we currently work on. Also, we want to extend our approach in order to support (semi-)automatic creation of safety arguments based on the proposed metamodel.

Acknowledgements. This work has been partially sponsored by the Austrian Ministry for Transport, Innovation and Technology (IKT der Zukunft, Project SALSA) and the Munich Center for Internet Research (MCIR). The author B. Gallina is financially supported by the ECSEL JU project AMASS (No 692474).

References

1. Ayoub, A., Kim, B., Lee, I., Sokolsky, O.: A Systematic Approach to Justifying Sufficient Confidence in Software Safety Arguments. In: Proceedings of Computer Safety, Reliability, and Security. vol. 7612 of Lecture Notes in Computer Science, pp. 305–316. Springer, Berlin, Heidelberg (2012)
2. Bennion, M., Habli, I.: A Candid Industrial Evaluation of Formal Software Verification Using Model Checking. In: Companion Proceedings of the 36th International Conference on Software Engineering. pp. 175–184. ACM, New York, NY, USA (2014)
3. Bloomfield, R.E., Bishop, P.G.: Safety and Assurance Cases: Past, Present and Possible Future - an Adelard Perspective. In: Making Systems Safer - Proceedings of the 18th Safety-Critical Systems Symposium. pp. 51–67. Springer, London (2010)
4. Bourdil, P.A., Dal Zilio, S., Jenn, E.: Integrating Model Checking in an Industrial Verification Process: a Structuring Approach (2016), https://hal.archives-ouvertes.fr/hal-01341701, working paper or preprint
5. Cârlan, C., Beyene, T.A., Ruess, H.: Integrated Formal Methods for Constructing Assurance Cases. In: Proceedings of International Symposium on Software Reliability Engineering Workshops. pp. 221–228. IEEE (2016)
6. Cârlan, C., Ratiu, D., Schätz, B.: On Using Results of Code-Level Bounded Model Checking in Assurance Cases. In: Proceedings of Computer Safety, Reliability, and Security Workshops. vol. 9923 of Lecture Notes in Computer Science, pp. 30–42. Springer, Cham (2016)
7. Denney, E., Pai, G.: Evidence Arguments for Using Formal Methods in Software Certification. In: Proceedings of International Symposium on Software Reliability Engineering Workshops. pp. 375–380. IEEE (2013)
8. Gallina, B.: A Model-Driven Safety Certification Method for Process Compliance. In: Proceedings of International Symposium on Software Reliability Engineering Workshops. pp. 204–209. IEEE (2014)
9. Gallina, B., Andrews, A.: Deriving Verification-Related Means of Compliance for a Model-Based Testing Process. In: Proceedings of IEEE/AIAA 35th Digital Avionics Systems Conference. pp. 1–6 (2016)
10. Gallina, B., Kashiyarandi, S., Zugsbratl, K., Geven, A.: Enabling Cross-Domain Reuse of Tool Qualification Certification Artefacts. In: Proceedings of Computer Safety, Reliability, and Security Workshops. vol. 8696 of Lecture Notes in Computer Science, pp. 255–266. Springer, Cham (2014)
11. Gallina, B., Pitchai, K.R., Lundqvist, K.: S-TunExSPEM: Towards an Extension of SPEM 2.0 to Model and Exchange Tunable Safety-Oriented Processes. In: Proceedings of the 11th International Conference on Software Engineering Research, Management and Applications. pp. 215–230. Springer SCI (2014)
12. Garfinkel, S.: History's worst software bugs (2005), http://archive.wired.com/software/coolapps/news/2005/11/69355?currentPage=all
13. Goodenough, J., Weinstock, C.B., Klein, A.Z.: Toward a Theory of Assurance Case Confidence. Tech. Rep. CMU/SEI-2012-TR-002, Software Engineering Institute, Pittsburgh, PA, USA (2012)

14. Graydon, G., Knight, J.: Process Synthesis in Assurance-Based Development of Dependable Systems. In: Proceedings of 8th European Dependable Computing Conference. pp. 75–84. IEEE (2010)
15. Habli, I., Kelly, T.: A Generic Goal-Based Certification Argument for the Justification of Formal Analysis. Electronic Notes in Theoretical Computer Science 238(4), 27–39 (2009)
16. Hawkins, R., Kelly, T.: A Structured Approach to Selecting and Justifying Software Safety Evidence. In: Proceedings of 5th International Conference on System Safety. pp. 1–6. IET (2010)
17. Höller, A., Kajtazovic, N., Rauter, T., Römer, K., Kreiner, C.: Evaluation of Diverse Compiling for Software-Fault Detection. In: Proceedings of the Design, Automation & Test in Europe Conference & Exhibition. pp. 531–536. IEEE (2015)
18. Holloway, C.M.: Explicate'78: Uncovering the Implicit Assurance Case in DO-178C. Tech. Rep. 20150009473, NASA Langley Research Center (2015)
19. Object Managment Group: Structured Assurance Case Metamodel - SACM, version 2.0 Beta. Tech. rep. (2016), http://www.omg.org/spec/SACM/2.0/Beta1/PDF/
20. Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., Ng, A.: Ros: An Open-Source Robot Operating System. In: Proceedings of Open-Source Software Workshop Int. Conf. Robotics and Automation. vol. 3. IEEE (2009)
21. RTCA: DO-178C, software considerations in airborne systems and equipment certification. RTCA & EUROCAE (2011)
22. RTCA: DO-333 formal methods supplement to DO-178C and DO-278A. RTCA & EUROCAE (2011)
23. Shiraishi, S., Mohan, V., Marimuthu, H.: Test Suites for Benchmarks of Static Analysis Tools. In: Proceedings of International Symposium on Software Reliability Engineering Workshops. pp. 12–15. IEEE (2015)
24. Weaver, R., McDermid, J., Kelly, T.: Software Safety Arguments: Towards a Systematic Categorisation of Evidence. In: Proceedings of the 20th International System Safety Conference. System Safety Society (2002)
25. Wei, C., Xiaohong, B., Tingdi, Z.: A Study on Compiler Selection in Safety-Critical Redundant System based on Airworthiness Requirement. Procedia Engineering 17, 497–504 (2011)