Building Heterogeneous Safety Cases for Automatically Generated Code

Nurlida Basir

Universiti Sains Islam Malaysia, 71800 Nilai, Negeri Sembilan, Malaysia

Ewen Denney

SGT/NASA Ames, Moffett Field, CA 94035, USA

Bernd Fischer

ECS, University of Southampton, Southampton, SO17 1BJ, UK

Safety cases provide a mechanism for representing evidence-based arguments that a system is acceptably safe to operate in its intended context. We show how we can automatically combine diverse types of information from heterogeneous sources into a single integrated safety case for a system implemented using automatically generated software. The core argument structure of the safety case is generated from a formal analysis of automatically generated code, based on automated theorem proving, and driven by a set of formal requirements and assumptions. This is then extended by separately specified auxiliary information giving contexts, assumptions, justifications, and constraints, or additional forms of evidence derived from other verification activities, such as testing. The resulting safety case thus combines formal and informal argumentation and makes explicit assumptions which would otherwise be left implicit.

Keywords: safety cases, model-based software development, automated code generation, formal proofs, formal analysis, automated theorem proving, V&V.

I. Introduction

Model-based development and automated code generation are increasingly used for actual production code, in particular in mathematical and engineering domains. For example, NASA's Project Constellation uses Real-Time Workshop for its Guidance, Navigation, and Control (GN&C) systems. However, since code generators are typically not qualified, there is no guarantee that their output is correct or even safe. In previous work, we have shown how safety and functional requirements can be formally verified for control software that has been generated automatically from Simulink models,[9] and how the results of this formal verification phase can be communicated in the form of a safety case.[4]

However, a comprehensive safety case must integrate additional extra-logical information into the core argument structure representing the purely formal reasoning, e.g., a justification that the formalizations of the properties that are verified correctly encode the requirements, or links to applicable standards and project documentation. This additional information can represent background knowledge that cannot be produced directly by the formal verification phase. It thus needs to be specified in the form of contexts, assumptions, justifications, and constraints. It can also represent additional forms of evidence derived from other verification activities, such as testing. In both cases, this information needs to be spliced into the core argument structure at the appropriate locations. The additional information can also represent knowledge about the system structure, e.g., its architecture, or tracing information between code and model. This structural information needs to be passed down to and processed by the formal verification phase, because it can directly influence the construction of the core argument of the safety case.

In this paper, we show how we can automatically combine these diverse types of information from heterogeneous sources into a single integrated safety case. The core argument structure of the safety case is generated from a formal analysis of the automatically generated code, based on logical annotation inference

and automated theorem proving. The analysis is driven by a set of formal requirements and assumptions on the system's output and input signals, respectively, but otherwise remains independent of the model structure. In particular, it analyzes the system structure on the code level to identify where the requirements are ultimately established, and so checks the model, providing independent assurance. The resulting argument structure is then extended by the auxiliary information, which is separately specified.

We have developed the overall structure of the heterogeneous safety case and instantiated it for the code generated for a Guidance, Navigation, and Control (GN&C) subsystem. The safety case thus provides a traceable safety argument that shows in particular where the code, documentation, verification and validation artifacts and the argument itself depend on any external assumptions. The resulting safety case explicitly highlights the claims, key safety requirements, and evidence that are required to trust both the software and its verification.

II. Background

A. Automated Code Assurance

Model-based design and automated code generation (or autocoding) are being used increasingly at NASA. They promise many benefits, including higher productivity, reduced turn-around times, increased portability, and elimination of manual coding errors.[8,21] There are many successful applications of both in-house custom generators for specific projects, and generic commercial generators. One of the most popular code generators within NASA is the MathWorks Real-Time Workshop[24] (with the add-on Embedded Coder), an automatic code generator that translates Simulink/Stateflow models into embeddable (and embedded) C code.

However, there still remain significant obstacles to more widespread adoption of code generators in safety-critical domains, principally, how the generated code should be assured. Since code generators are typically not qualified, there is no guarantee that their output is correct, and consequently the generated code still needs to be fully tested and certified. Ideally, the code generator, itself, should be qualified. However, this is a non-trivial and expensive process, and is therefore rarely done. Moreover, the qualification is only specific to the use of the generator within a given project, and needs to be redone for every project and for every version of the tool. Also, even if a code generator is generally trusted, user-specific modifications and configurations necessitate that V&V be carried out on the generated code.[12]

Users also need to be sure that the code implements the model, that the code generator is correctly used and configured, that the target adaptations are correct, that the generated code meets high-level safety requirements, that it is integrated with legacy code, etc. There are also often concerns with the understandability of the generated code. Some understanding of why the code is safe, therefore, helps the larger certification process. Automated support for V&V that is integrated with the generator can address some of these complexity concerns. Furthermore, certification requires more than black box verification of selected properties, otherwise trust in one tool (the generator) is simply replaced with trust in another (the verifier). Automated code generation, therefore, presents a number of challenges to software processes and, in particular, to V&V, and this leads to risk. The AutoCert tool we describe here, and its safety case extension, mitigates some of that risk.

B. AutoCert

AutoCert[10] is a generator plug-in to support the subsequent certification of the code created by the Real-Time Workshop[24] code generator. It supports certification by formally verifying that the generated code complies with a range of mathematically specified requirements and is free of certain safety violations. AutoCert certifies every generated program individually, rather than the generator itself: given a set of formal assumptions (e.g., constraints on input signals) and requirements (e.g., constraints on output signals), it formally verifies that the generated code complies with the specified requirements by constructing an independently verifiable certificate. This enables high-level assurance about the safety and reliability of the code without excessive manual verification and validation effort.

AutoCert follows the Hoare logic approach to verification; in particular, it carries out a symbolic analysis of the generated source code in order to prove properties about the code, rather than the model. It produces assurance evidence which can be checked either by machines (i.e., proof checking) or by humans (i.e., code reviews). The key technical idea of AutoCert is to exploit the idiomatic nature of auto-generated code and to use annotation templates in order to automatically infer logical annotations, i.e., assertions of

program properties at key locations in the code. Annotations are crucial in order to allow the automatic formal verification of the safety properties without requiring access to the internals of the code generator, as well as making a precise analysis possible. The annotations are used to generate verification conditions (VCs), which are then proved by an automated theorem prover. The annotation templates are used to produce the annotated program, while the annotation inference is used during the safety case generation process. Note that only the VCs and proofs are used as evidence in the argument, but not the annotation templates. Instead, templates can appear as models (in the GSN sense) attached to strategies.

During the course of verification, AutoCert records various facts, such as the locations of variable definitions and uses, which are later used in the safety case generation process. The AutoCert approach is independent of the particular generator used, and need only be customized by the appropriate set of patterns. The approach also shifts the trust burden from the program to the certification system: instead of having to trust an arbitrary program to be safe, users only have to trust the certifier to be correct.

III. Case Study: Guidance, Navigation and Control (GN&C) Subsystem

We illustrate our work using excerpts that explain the verification of several functional requirements for an attitude module of a spacecraft GN&C system which has been auto-generated using Real-Time Workshop. GN&C is a necessary component of every spacecraft. The GN&C domain is challenging from a verification perspective due to its complex and mathematical nature.[11,22,29] We just describe the model at a high level sufficient to understand typical requirements. The attitude subsystem takes several input signals, representing various physical quantities, and computes output signals representing other quantities, such as Mach number, angular velocity, position in the Earth-Centered Inertial frame etc. Signals are generally represented as floats or quaternions and have an associated physical unit and/or frame of reference. At the model level, the transformations of coordinate frames are usually done by converting quaternions to direction cosine matrices (DCMs), applying some matrix algebra, and then converting back to quaternions. Other computations are defined in terms of the relevant physical equations. Units and frames are usually not explicit in the model or the code, and instead are expressed informally in comments and identifier names.

The attitude subsystem (ATT) is comprised of three sub-subsystems or components, a decision logic that computes a status value irrelevant to the the requirements we consider here, a frame conversion (FC), and a state determination (SD). FC first converts the frames of reference the incoming signals from a vehicle-based coordinate system to an earth-based coordinate system. The transformations of the coordinate systems are usually done by converting quaternions to direction cosine matrices (DCMs), applying some matrix algebra, and then converting them back to quaternions. SD then performs the calculations to determine the vehicle state (position, attitude, attitude rate, etc.) from these signals. It is defined in terms of the relevant physical equations. Note that there are no individual blocks within the Navigation subsystem, but only within the components and thus all computation happens there.

In our work, we combine information provided by AutoCert together with other background information of the GN&C system (mission documents such as Concepts of Operations (ConOps), scientific glossaries, the software requirements specification (SRS), environmental constraints and assumptions on the system, as well as documents that describe the architecture and model of the system) in order to construct the safety case. This additional background information cannot be produced directly by the formal verification phase. The combination of the formal verification information and additional background information of the system is necessary in the construction of the core argument of the safety case.

IV. Generating Safety Cases

A. Safety Case Purposes

In principle, formal methods can offer a strong form of evidence in system safety.[25] However, formal evidence by itself is inadequate to justify why and how the formal analysis achieves the certification objectives. Therefore, a valid justification is essential to support the assurance claim provided by the formal evidence. Safety cases are one of the most widely used techniques to communicate the relationship between evidence and objectives. A safety case is defined in UK Defence Standard 00-56[25] as a *"structured argument, supported by a body of evidence that provides a compelling, comprehensible and valid case that a system is safe for a given application in a given operating environment"*.

In our work, we construct heterogenous safety cases that make explicit the formal and informal reasoning principles, and reveal the top-level assumptions and external dependencies that must be taken into account in demonstrating software safety. They also provide information about why the generated code can be assumed to be sufficiently correct. They can thus be thought of as *"structured reading guide"* for the safety proofs and act as a traceable route to the safety requirements, safety claims and evidence that are required to show correctness of the generated code. Their core argument structure is generated from a formal analysis of the automatically generated code. Safety cases help in providing information about why the code can be assumed to be sufficiently safe and correct, by analyzing the system structure on the code level to identify where the requirements are ultimately established, and so check the model, providing independent assurance. Additional forms of evidence that derived from other verification activities, such as testing are also described in the core argument structure at the appropriate locations. We use the Goal Structuring Notation (GSN)[20] as graphical argumentation notation to explicitly represent the logical flow and linkage between safety argument elements.

B. Heterogenous Safety Case Structure

Demonstrating the correctness of large and complex software-intensive systems in certification requires marshalling large amounts of diverse information, including mission documents that describe summary of NASA's program and the overall mission scenario, detail requirements of the system to be developed, system architecture, Matlab models of the system to be developed to address problems associated with designing complex control systems, source code of the system and formal V&V information such as verification conditions, proof framework, formal proofs and testing results. An integration of these diverse artifacts is essential to explicitly represent the rationale underlying the certification process and to establish trust and confidence on the certification provided. The following subsections describe how we combine these diverse types of information from heterogenous sources into a single integrated safety case. However, we only provide a simplified overview of this safety case and concentrate on its generic structure.

1. Certification Files

In order to certify a program, AutoCert reads a *certification file* (see Figure 1 for the syntax), which specifies the assumptions and requirements for the code under verification together with the non-formal information that will be woven into the formally derived core of the safety case. A certification file consists of a set of assumptions, requirements, and associated evidence.

```
CertFile    ::= Assumption* Requirement* Evidence*

Formula     ::= Signal::Type | Signal :: bus(Type*) | Signal1=Signal2 | Formula => Formula
              | Formula /\ Formula

Requirement ::= requirement(Formula, Extension*)

Assumption  ::= assumption(Formula, Extension*)

Evidence    ::= evidence(EntityType, EntityName, Extension*)

Extension   ::= text(Text) | context(Text) | justification(Text) | constraint(Text)
              | model(Text) | type(VerifType) | id(Id)

EntityType  ::= axiom | vc | libfun | testdata | doc | mapping | formalization | ...

VerifType   ::= inspection | testing | prover(P) | ...

Mapping     ::= mapping(ModelEntityName, CodeEntityName)
```

Figure 1. Syntax of Certification Files

Assumptions are properties of input variables (corresponding to signals and buses in the model) and requirements are properties of output variables. For each assumption and requirement, the certification

engineer can specify additional non-formal information. This includes a unique external identifier (e.g., `id('4.1.1')`) and a natural language textual representation of the formula, in addition to justification and context information that will be attached to the corresponding assumption and goal nodes; if none is given, it constructs a default natural language representation of the underlying formula. For example:

```
assumption('ATT_Mode' > 0, [id('4.1.1'),
 text('Attitude subsystem is triggered'),
 justification('Only consider the nominal case where the attitude subsystem is active'),
 context('The trigger is computed by Check_ATT_Mode from other inputs')]).

assumption(7_29212e_minus_05 :: ang_vel(eci), [
  justification('Five decimal places is sufficiently accurate for angular velocity')]).

requirement('AttECIToBody':: quat(eci,body), [id('SRS Rqmt 3.2'),
  text('The software shall convert the nav message data to an
        Earth-Centered Inertial coordinate system'),
  context('Hazard Analysis'), context('Project documentation on coordinate systems')]).
```

Evidence consists of justifying information for the leaf nodes of the safety case, namely axioms, verification conditions, and library functions. For example:

```
evidence(formalization, 'SRS Rqmt 3.2', [id('http://.../3.2'),
  type(doc), text('Documentation of the domain theory')]).

evidence(axiom, transpose_matrix, [id('testdata/axioms/transpose_matrix_testdata'),
  text('Frames for matrix transpose'), type(testing)]).

evidence(libfun, 'compute_dcm_ned_body_func, [id('testdata/libfuncs/dcm_ned_body_testdata')
  text('Function spec testing'), type(testing)]).
```

Note that axioms can be tested with respect to a reference implementation using the techniques of Ahn and Denney.[2] We can also specify "second-order" evidence, which refers to manually inspected supporting evidence for machine-produced evidence, for example:

```
evidence(testdata, 'testdata/libfuncs/dcm_ned_body_testdata', [
  type(inspection), text('Inspection by SME')]).

evidence(vc, 'vc0013', [type(inspection), text('Manually proven by ...')]).

evidence(doc, srs, [id('mission/docs/software/srs.doc')]).
```

Since we are working with a formal, logic-based analysis framework, we need to formalize the assumptions and requirements using a domain theory. The expression $X :: T$ denotes that X has the unit or frame T, depending on the kind of T. *ECI* and *Body* are constants denoting frames, while *quat* and *vel* are functions denoting transformations of or quantities in those frames. An important function of the safety cases is to demonstrate that this formalization is adequate, which has two aspects. First, we show that the formula itself is adequate, by giving additional evidence in the certification file that points to the documentation of the domain theory (see above). Second, we need to explicitly specify the mapping between the signal names used in the model and the corresponding variable names used in the source code, which cannot be recovered by our analysis but must be given externally. Here, the certification file contains the mapping information given, which is used in the analysis, and additional evidence showing that this has also been checked by a reviewer. For example:

```
mapping('AttECIToBody', 'rtw_ecitobody').

evidence(mapping, 'AttECIToBody', [
  type(inspection), text('Name mapping information inspected on ... by ...')]).
```

2. Architecture Recovery

In order to certify the requirements on a system, and to build a comprehensible safety case, we need to know where in the system they are established, and which parts of the system contribute to them. However,

a purely model-oriented view is too simplistic. First, without looking inside the component models it is not clear whether a requirement is indeed established within a component, or simply passed through, and which of the component's input signals (if any), or more precisely which assumptions on them, are used in establishing the requirement. Simply expanding the component models destroys the hierarchical structure of the system. Second, and more importantly, the safety of the system ultimately depends on the safety of the code rather than the model, but because we cannot trust the code generator to translate the model correctly we cannot derive any trust from the model.

Instead, we analyze the code and recover the slice of the system architecture that is relevant to a given safety requirement. The key to obtaining precise architecture slices is to record when the analysis enters resp. leaves a component, and then to identify situations in which the control flow just passes through a component, without encountering a definition. In these cases, we can ignore the component altogether. We then assemble the slices from the signals involved and from the retained component.

For each top-level system requirement specified in the certification file, AutoCert thus identifies which components contribute to it. For each of these components it identifies not only the guarantees that the component needs to provide in order to satisfy the given requirement, but also the assumptions that it in turn makes on the system or other components. These become subordinate requirements to the original safety requirement, reflecting the hierarchical model structure. By regrouping the analysis results by component rather than by original safety requirement, we thus obtain full component interfaces. They give a complete functional specification of the components, including all assumptions, as far as required by the given system-level safety requirements. The interfaces also serve as starting point for verifying the components independently, hence allowing a compositional (and therefore scalable) verification.

The recovery of the system architecture is completed by identifying the signals that occur in the derived requirements and through which the components are thus connected. The recovered system architecture and requirements hierarchy then constitute a core safety argument: the system satisfies the safety requirements if the components satisfy their respective interfaces, and the requirements for the signals hold. This argument serves as blueprint for the safety case. In addition, the derived component interfaces serve as starting points for the construction of independent safety cases for the components, yielding a hierarchy of safety cases that is aligned with the system's hierarchy of models.

3. Top-Level Structure of the Safety Case: Tier I

The safety case[a] (see Figure 2) starts with the top-level safety goal (i.e., to show that the software satisfies all given requirements) and shows how this is achieved by an explicit argument based on the partial correctness of the generated software wrt. the requirements given in the certification file. The argument stresses the meaning of the Hoare-style framework using specific proof rules but its structure remains independent of the given generator and software. The strategy is predicated on two assumptions, first, that the proof of correctness ensures that the functional requirements and safety requirements are established and maintained during execution, and second, that process scheduling and idealized treatment of floating point arithmetic do not compromise the safety claim. The first subgoal, i.e., a formal proof that the software satisfies all given requirements, is further elaborated in the lower tiers of the safety case. The second subgoal is to show that the safety policy adequately represents the safety property. However, this subgoal is not elaborated further here but leads to a complementary safety case for the safety logic.

Contexts explain additional information for the argument to be understood, e.g., the generator and model that are used to generate the software, and describe the software safety context i.e., in terms of functional and safety requirement. Constraints outline limitations of the approach, in particular, the fact that certification works on an intermediate representation of the source code and only shows partial correctness proof (i.e., no termination) without any real-time reasoning. Hyperlinks refer to additional evidence in the form of documents, containing, for example, the model from which the software has been generated and its configuration set. This information is specific to the given software. It is is either collected during the formal analysis (e.g., the list of files), or specified as evidence in the certification files. Since the top-level structure of the safety case is completely generic, the safety case generator can splice in the specific information at the right locations.

The safety case continues by arguing over the correctness of the software w.r.t. each individual requirement. Again, additional information that is required for the strategy to be understood and valid is identified

[a]We have edited the generated safety cases for ease of presentation. In particular, we have split the single large case into separate "tiers", and made several simplifications.

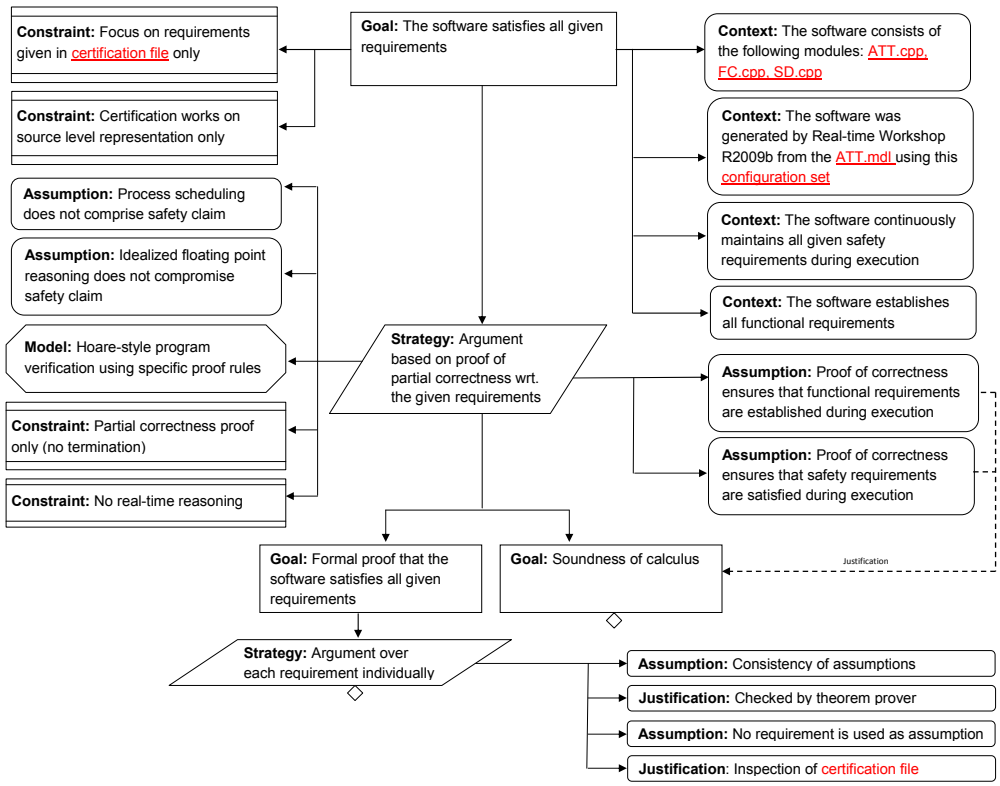

Figure 2. Top Level of the Safety Case (Tier I): Arguing over Software Safety

and explained. This concerns the independent validity of the safety requirements and the logical consistency of the assumptions. We assume that no safety requirement is available for use as a (logical) assumption in the safety proofs, which prevents vacuous proofs based on mutually recursive dependencies between requirements and assumptions. We further assume that the given and derived assumptions together are consistent, again to prevent vacuous proofs. An assumption can be justified by an explicit justification (e.g., the consistency can be checked by theorem prover), which is specified in the certification file.

4. Architecture-Level Safety Case: Tier II

As a result of the last strategy in the previous tier, we get as many subgoals as there are safety requirements. Here, we focus on the subgoal corresponding to the requirement given in the certification file, i.e., to show that the signal AttECIToBody is a quaternion representing the specified frame transformation. Additional context information also specified in the certification file is attached to the node.

The next step of the argument transitions from the informal level to a formalized safety requirement. This step helps in showing that the formal verification runs over the correct requirement, based on the right formula and variable, and thus provides a relevant proof of the program. We use an explicit strategy to describe this transition, which spawns three subgoals. The first subgoal demonstrates that the formal proof is based on an appropriate formalization of the requirement, and the safety case points to the documentation of the logical domain theory as evidence of this; this evidence *must* be given in the certification file to complete the argument, and we could even stop the safety case construction with an error if it is missing.

The second subgoal "glues together" model and code levels, which allows us to build a safety case for the model based on the analysis of the code. We need to show the mapping between the signal names used in the model and the corresponding variable names, which cannot be recovered by our analysis but must be given in the certification file. Here, the safety case points to the mapping information, and to the fact that it has been checked by a reviewer, also given as evidence in the certification file. In addition, at this goal we also have to show the mapping between the model and code files, and in particular, in which code file

the formalized property has to be shown. In our example, this is straightforward and the information can be collected by the analysis, but for larger systems this localization might need more evidence that can be specified in the certification file.

We can then construct the final subgoal of our strategy, which shows that the fully formalized safety requirement AttECIToBody :: quat(eci, body) holds after execution of the code in ATT.cpp. This is proven formally by a compositional verification based on the system architecture, or more precisely, on the recovered system architecture slices. The safety case shows how the system level requirements are broken down into the component level requirements, i.e., properties of the part of the system that is relevant to satisfy the requirement. The strategy is based on the assumption that the formal analysis has identified all relevant components and signals. For each component, we need to show that it satisfies the safety requirements specified in its interface (subgoal (C1)). This induces a further assumption on the strategy, namely that the interface is strong enough to show the requirement (FR1). For each variable representing a signal, we need to show that it satisfies the safety requirements derived by the analysis (i.e., subgoals (S1) and (I1) to (I_n)). This guarantees that the components' assumptions are met. These subgoals are delayed here, to keep the safety case compact. Here, we do not distinguish (other than in the numbering) between internal assumptions (e.g., on rtw_bodytonav), which have been identified as subgoals in the system-level safety case, and external assumptions to the entire system. However, not all assumptions are used for all requirements, so we use an explicit strategy to argue only using the minimal set of external (i.e., on the system's input signals) assumptions.

5. Code-Level Safety Case: Tier III

In the next tier, we argue about the safety of the components w.r.t. their identified interfaces. This tier also argues about a set of requirements, but there are two significant differences to the architectural-level safety case. First, the component-level requirements are already formalized, due to the use of the formal analysis, so that we do not need to argue about the safety of the formalization and localization any more. Second, the argument will generally go down to the level of the generated code, with the proofs of the VCs as evidence; obviously, however, another layer of hierarchy is introduced if a component contains further components.

This tier (see Figure 3) finally transitions from the safety argument to a program correctness proof, using a Hoare-style argument over all relevant occurrences of the variable. The structure of this Hoare-style argument is determined by the structure of the program. In this case, it leads to a single subgoal, proving that the safety requirement holds at the given source location.

The final elements of the safety case argue that his is the case, using a strategy based on establishing and maintaining appropriate conditions. This directly reflects the course the annotation inference has taken through the code. The first subgoal is thus to show that the sufficient conditions are established on all paths leading to the current location, using an argument over all definition locations. Here, the model for the subgoal corresponds to the pattern that was applied during annotation inference to identify the definition. Each definition (e.g., nav_dcm_22 is established with frame DCM(ned, body) at line 288) thus leads to corresponding subgoals i.e., the library function (e.g., compute_dcm_ned_body) is correctly specified and VCs are proven. The validity of each subgoal should be explicitly justified. Here, the correctness of library functions are justified by testing results. Any established methods or documents (e.g., reference implementation document) that can be used to support the testing processes are described as model that link to the strategy. The validity of the testing results that have been inspected by experts (e.g., by the Subject Matter Expert (SME)) is specified as direct evidence to the goal. While the validity of the construction of the VCs depends on the soundness of the rules of the VCG, the simplifier, and the definition of the safety policy, while the correspondence to program locations is based on tracing information added by the VCG and retained during the certification process.

The second subgoal of the top-level strategy is to show that the established conditions are maintained on all paths. This proceeds accordingly and the VCs demonstrate that the conditions are maintained. The final subgoal is then to show that the conditions from each path imply the required property. This can again lead to any number of VCs. If (and only if) all VCs can be shown to hold, then the property holds for the entire program. The evidence on soundness of VCs are shown in Tier IV.

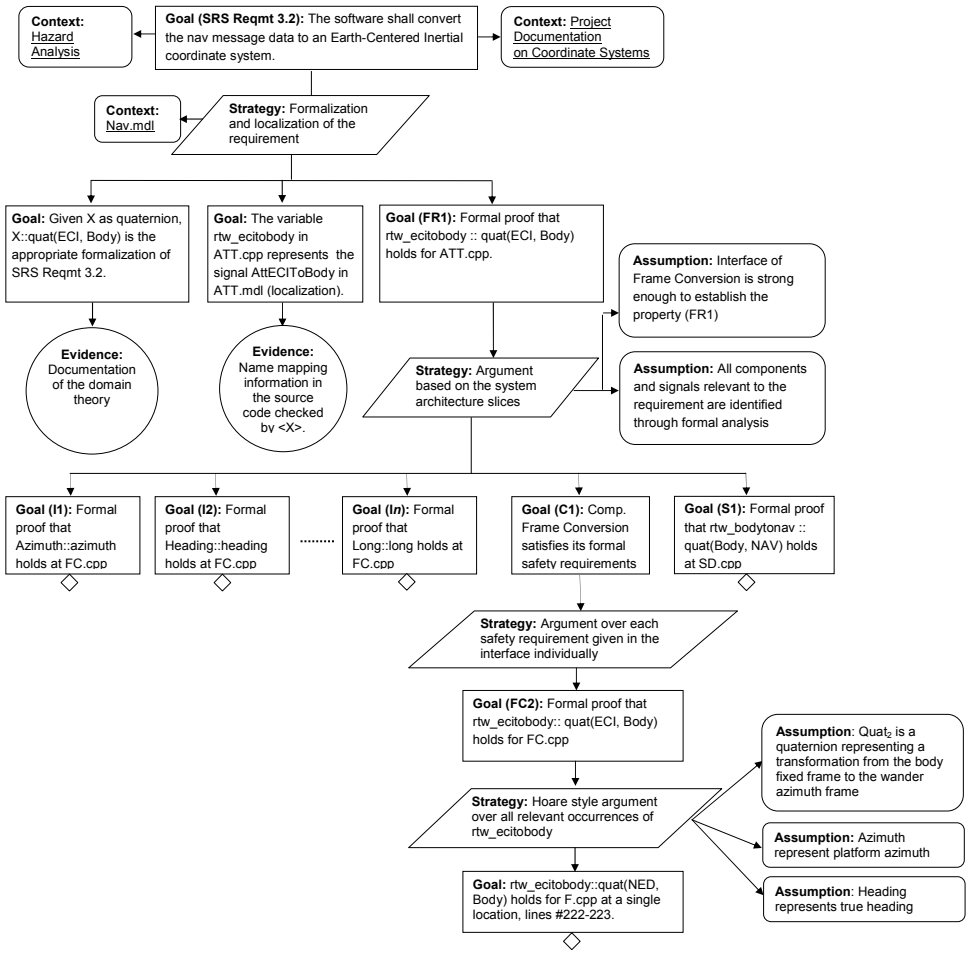

Figure 3. Architectural Level of the Safety Case (Tier II): Arguing over Requirements and Architecture

6. From Formal Proofs to Safety Arguments: Tier IV

In formal verification, automated theorem provers are typically used to demonstrate the validity of theorems. The soundness, correct configuration and installation of the prover should also be justified in order to establish trust on the proofs. Tier IV (see Figure 5) of the safety case presents the underlying argumentation structure and top-level assumptions of the formal proofs found by the automated theorem provers. We applied our approach to proofs found by the SPASS theorem prover. We also highlight the use of the external certification assumptions in order to check the validity of their use in deriving the proofs. The argument starts with the generated VC to be proven as the top goal and follows the deductive reasoning down into subgoals, using the applied inference rules as strategies to derive the goals, and has the given axioms or assumptions at the leaves. The use of external certification hypotheses (e.g., vs_lat is a geodetic latitude and hirate signal is within range) that have been formulated in isolation by the safety engineer are tracked in the safety case. By revealing all external and internal hypotheses, the validity of their use in deriving the proof can be checked easily. The soundness of the formal proofs rests on the validity of the axioms which are used. This can be shown by testing them with respect to a reference implementation.[2] The reference implementation that is used for this is specified in the form of a model and should, itself, be inspected by a SME.

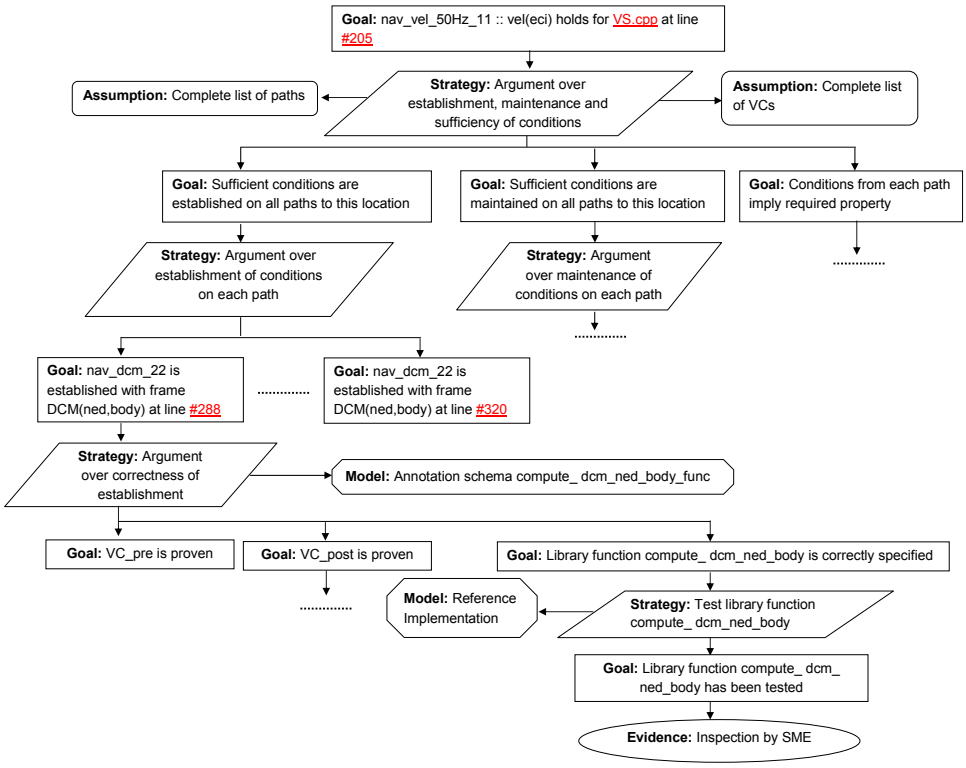

Figure 4. Code Level of Safety Case (Tier III): Arguing over Code Structure

C. Safety Case Construction

The development and acceptance of safety cases is a key element of the safety regulation in many safety-critical sectors. Apparently, most safety cases are constructed manually as no advance tool is available to support an automatic safety case construction. Most safety case construction tools only provide basic drawing support. However, the manual construction of a safety case is impractical especially when we are dealing with large amounts of artifacts and iterative software development. It is not only a time consuming and expensive process, but it might also slow down the application of the safety case.

To automatically construct the safety case, we integrate AutoCert with Adelard's ASCE tool.[1] We convert the information derived by the annotation inference into an XML format, and merge any context, evidence, justification, and text information explicitly specified in the certification file into this. We use the identifiers and the entity types to determine where in the safety case this information must be added. Some information is fixed by our verification technique (e.g., that axioms are tested with respect to reference implementations), whereas other aspects follow from the specific details of the verification itself (e.g., the specific inference templates which are used during the verification).

This XML file thus contains all relevant information that is required for the construction of the heterogenous safety case. We then use a set of XSLT transformations to construct a second XML file in the GSN-XML format that logically represents the safety case. The file format was designed so that the derived safety cases can be easily be adapted to different tools or applications. Finally, to present the resulting safety case graphically, we use a Java program to layout the logical information which involved some mathematical calculations in positioning the argument and to convert it into the standard Adelard ASCE file format.

V. Related Work

The development and acceptance of a safety argument or safety case is now a key element of safety regulation in many safety-critical sectors.[25,30] There has been work on demonstrating how software fits into the safety of an overall system. For example, Weaver[31] in his thesis presents arguments that reflect the

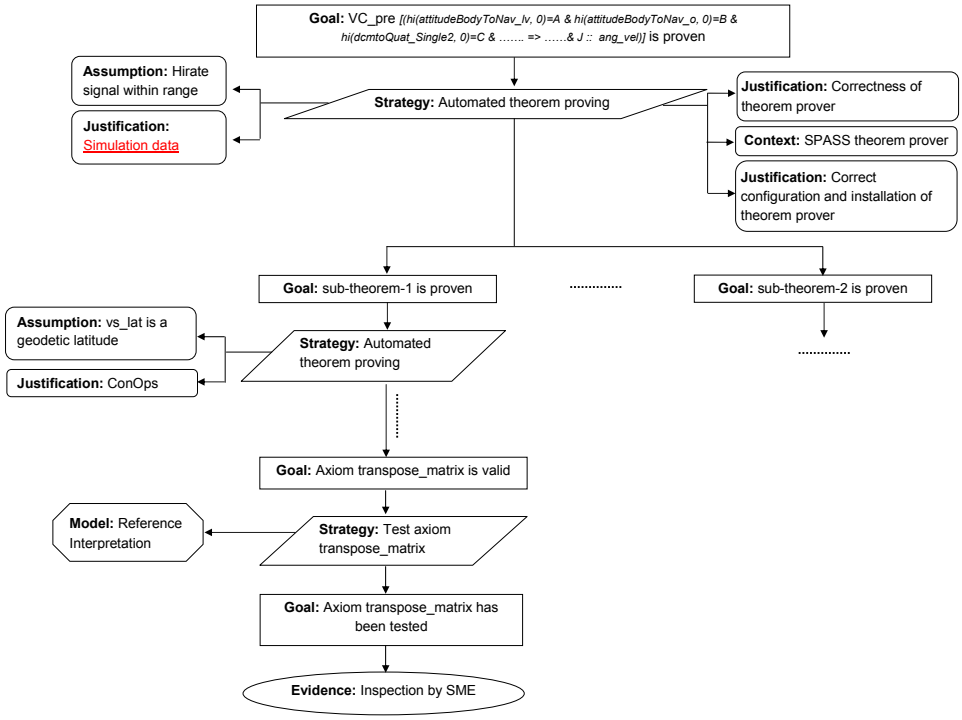

Figure 5. Derived Safety Case (Tier IV): Arguing over Proofs

contribution of software to a safety-critical system and Reinhardt[26] looks at the safe application of the C++ programming language in safety-critical systems. Our approach works in the same context, i.e., presents arguments over the safety of the generated program with respect to the given requirements for its use in safety-critical systems. Audsley et al.[3] present arguments over the correctness of the specification mapping, i.e., a translation from the system specifications into a model and subsequently into a code. This is similar to our approach of showing the correct formalization and localization of the requirement.

With the increasing use of model-based development in safety-critical applications, the integration of safety cases into such approaches has become an important research topic. For example, Chen et al.[7] describe an integration of model-based engineering with safety analysis and safety cases to help in assessing decisions in the system design of automotive embedded systems. Similarly, Wu[32] introduces a framework to facilitate safe architectural design in safety-critical applications. Hause and Thom[18] describe how SysML and UML can be used to model system requirements and how the safety requirements and other system elements identified in the system design were used to construct the safety case. The focus in their research, however, is on extending the modeling framework to represent safety cases using the applied notation. In contrast, in our work, we construct a safety case that argues along the hierarchical structure of systems in model-based design and show how the hierarchy of requirements is aligned with the hierarchical model structure. Rushby[27] also uses automated theorem proving technology (based on the Yices SMT solver) to make a safety argument, but does not construct a detailed safety case. Moreover, his analysis starts with a manually constructed logic-based model of the system, where the connection to the underlying code remains unclear while our approach works directly on the code. Takeyama[28] uses Agda as an underlying framework for the construction of well-formed safety cases.

In order to demonstrate a compelling argument on software safety assurance, Hawkins and Kelly[19] provide a framework for justifying the arguments and evidence required to demonstrate sufficient assurance in software. Littlewood and Wright[23] state that the probability of a claim, which has been shown by a formal proof, being false, is very low, when the assumptions and evidence are valid. As pointed out by Littlewood and Wright,[23] we believe substantial confidence can be obtained if the validity of the underlying proof construction can be shown. Galloway et al.[14] present arguments for technology substitution, i.e., argue over

substitution of testing with a proof-based verification technique in the context of the certification standards such as DO-178B,[13] while Habli and Kelly[15–17] carried out research on constructing a safety argument to facilitate the justification and presentation of formal analysis in supporting testing techniques as presented in software standards. Similarly, our work justifies the application of a formal program analysis method in providing assurance for the software safety and the use formal proofs as an evidence.

However, all of these works[14–17] remain completely generic and do not take actual code into account. Our work in contrast focuses on constructing a defensible argument for how specific code complies with specific requirements (i.e., safety requirements and safety properties) based on the evidence (i.e., formal proofs and other V&V artifacts) available.

VI. Conclusions

Demonstrating the correctness of large and complex software-intensive systems requires marshalling large amounts of diverse information, including requirement documents, architecture diagrams, models of the system, source code, and V&V artifacts such as verification conditions, formal proofs, testing results, and software inspections. An integration of these diverse artifacts is essential to explicitly represent the rationale underlying the certification process and, thus, establish trust and confidence on the assurance provided.

In this paper, we have described the overall structure of a specific application of a heterogenous safety case. We have shown how we can systematically combine diverse types of information from heterogeneous sources into a single integrated safety case. The core argument structure of the safety case is generated from a formal analysis of automatically generated code, based on automated theorem proving, and driven by a set of formal requirements and assumptions. This is then extended by separately specified auxiliary information giving contexts, assumptions, justifications, and constraints, or additional forms of evidence derived from other verification activities. In previous work, we have automatically constructed property-oriented safety cases,[4] requirement-oriented and architecture-oriented safety cases,[6] and proof-oriented safety cases.[5] The aim of our work is to promote the use of formal mathematical arguments in achieving confidence in software safety. We also hope it will increase confidence in the use of formal methods as well as in the use of code generators in safety-critical applications.

Acknowledgments

This material is based upon work supported by NASA under contract NNA10DE83C.

References

[1]ASCE home page, 2010.

[2]Ki Yung Ahn and Ewen Denney. Testing first-order logic axioms in program verification. In *4th International Conference on Tests and Proofs (TAP 2010)*, Malaga, Spain, July 2010.

[3]N. Audsley, I. Bate, and S. Crook-Dawkins. Automatic Code Generation for Airborne Systems. In *Proceedings of the IEEE Aerospace Conference*, volume 4, pages 8–15. IEEE, 2003.

[4]Nurlida Basir, Ewen Denney, and Bernd Fischer. Constructing a safety case for automatically generated code from formal program verification information. In *The 27th International Conference on Computer Safety, Reliability and Security (SafeComp '08)*, Newcastle, England, 2008.

[5]Nurlida Basir, Ewen Denney, and Bernd Fischer. Deriving safety cases from automatically constructed proofs. In *The 4th IET International Conference on System Safety*, London, England, 2009.

[6]Nurlida Basir, Ewen Denney, and Bernd Fischer. Deriving safety cases for hierarchical structure in model-based development. In *The 29th International Conference on Computer Safety, Reliability and Security (SafeComp '10)*, Vienna, Austria, 2010.

[7]D.-J. Chen, R. Johansson, H. Lönn, Y. Papadopoulos, A. Sandberg F. Törner, and M. Törngren. Modelling Support for Design of Safety-critical Automotive Embedded Systems. In *Proceedings of the 29th International Conference on Computer Safety, Reliability and Security (SAFECOMP'08)*, volume LNCS 5219, pages 72–85. Springer, 2008.

[8]K. Czarnecki and U.W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[9]E. Denney and B. Fischer. A Generic Annotation Inference Algorithm for the Safety Certification of Automatically Generated Code. In *Proceedings of the Generative Programming and Component Engineering (GPCE'06)*, pages 121–130. ACM Press, 2006.

[10]E. Denney and S. Trac. A Software Safety Certification Tool for Automatically Generated Guidance, Navigation and Control Code. In *IEEE Aerospace Conference*, Big Sky, MT, USA, 2008.

[11]J. Diebel. Representing Attitude: Euler Angles, Unit Quaternions and Rotation Vectors, October 2006.

[12]Tom Erkkinen. High-integrity production code generation. In *Proceedings of AIAA GN&C Conference*, 2003.

[13]EUROCAE (European Organisation for Civil Aviation Equipment). ED-12B/DO-178B: Software Considerations in Airborne Systems and Equipment Certification, 1994.

[14]A. Galloway, R.F. Paige, N.J. Tudor, R.A. Weaver, I. Toyn, and J.A. McDermid. Proof vs Testing in the Context of Safety Standards. In *Proceedings of the 24th Digital Avionics Systems Conference (DASC'05)*, volume 2, page 14, 2005.

[15]I. Habli and T.P. Kelly. Process and Product Certification Arguments: Getting the Balance Right. *ACM SIGBED Review*, 3(4):1–8, 2006.

[16]I. Habli and T.P. Kelly. Achieving Integrated Process and Product Safety Arguments. In *Proceedings of the 15th Safety Critical Systems Symposium (SSS'07)*, pages 55–68, Bristol, UK, 2007. Springer.

[17]I. Habli and T.P. Kelly. A Generic Goal-Based Certification Argument for the Justification of Formal Analysis. *Electronic Notes Theoratical Computing Science*, 238(4):27–39, 2009.

[18]M.C. Hause and F. Thom. Integrated Safety Strategy to Model Driven Development with SysML. In *Proceedings of the 2nd IET International Conference on System Safety 2007*, volume CP532, pages 124–129, 2007.

[19]R.D. Hawkins and T.P. Kelly. Software Safety Assurance — What is Sufficient? In *Proceedings of the 4th IET International Conference on System Safety*, London, UK, 2009.

[20]Tim Kelly. *Arguing Safety: A Systematic Approach to Managing Safety Cases*. PhD thesis, University of York, 1998.

[21]Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained, The Model Driven Architecture: Practice and Promise*. Addison-Wesley, Reading, Mass., 2003.

[22]J.B. Kuipers. *Quaternions and Rotation Sequences*. Princeton University Press, 1999.

[23]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. *IEEE Transactions Software Engineering*, 33(5):347–365, 2007.

[24]MathWorks. Real-Time Workshop home page. `http://www.mathworks.com/products/rtw`.

[25]UK Ministry of Defence. *00-56 Safety Management Requirements for Defence Systems*, June 2007.

[26]D.W. Reinhardt. Use of the C++ Programming Language in Safety Critical Systems. MSc thesis. University of York, 2004.

[27]J. Rushby. A Safety-Case Approach for Certifying Adaptive Systems. *In:AIAA Infotech@Aerospace Conference*, 2009.

[28]Makoto Takeyama. A note on D-Cases as proofs as programs. Technical report, National Institute of Advanced Industrial Science and Technology, Osaka, Japan, 2010. AIST-PS-2010-007.

[29]D.A. Vallado. *Fundamentals of Astrodynamics and Applications*. Kluwer Academic Publishers, 2nd edition edition, 2001.

[30]J. Wang. Offshore Safety Case Approach and Formal Safety Assessment of Ships. *Journal of Safety Research*, 33(1):81–115, 2002.

[31]R.A. Weaver. *The Safety of Software Constructing and Assuring Arguments*. PhD thesis, University of York, 2003.

[32]W. Wu. *Architectural Reasoning for Safety-Critical Software Applications*. PhD thesis, University of York, 2007.