Evidence Arguments for Using Formal Methods in Software Certification

Ewen Denney and Ganesh Pai
SGT / NASA Ames Research Center
Moffett Field, CA 94035, USA
{ewen.denney, ganesh.pai}@nasa.gov

***Abstract*—We describe a generic approach for automatically integrating the output generated from a formal method/tool into a software safety assurance case, as an *evidence argument*, by (a) encoding the underlying reasoning as a safety case pattern, and (b) instantiating it using the data produced from the method/tool. We believe this approach not only improves the trustworthiness of the evidence generated from a formal method/tool, by explicitly presenting the reasoning and mechanisms underlying its genesis, but also provides a way to gauge the suitability of the evidence in the context of the wider assurance case. We illustrate our work by application to a real example–an unmanned aircraft system– where we invoke a formal code analysis tool from its autopilot software safety case, automatically transform the verification output into an evidence argument, and then integrate it into the former.**

***Index Terms*—Safety cases, Safety case patterns, Formal methods, Argumentation, Software certification.**

I. Introduction

In the *prescriptive safety certification* regime for software, assurance largely involves demonstrating that software will function as intended, achieved by showing (to a regulatory authority) that it complies with a number of process objectives set forth in the certification guidance documents, e.g., DO-178C [1]. The intention is to provide a level of assurance proportional to the safety-criticality of software, and the objectives to be met can include detailed processes to be followed and/or specific techniques to be used.

Formal methods, and the corresponding tools, provide a means to rigorously establish that software meets certain properties with respect to its specification, e.g., correctness. As far as safety is concerned, however, it is important to reason about software considerations in the *system context* to assure, for example, that the properties against which software has been certified do not adversely affect system safety. Safety cases, or more generally, assurance cases have emerged as a way to reason about both system safety and software considerations for system safety, where the focus is on *product-specific evidence* and *argumentation*.

Assurance cases link evidence (such as that produced by applying the prescribed verification processes, methods, and tools), to the stated assurance claims through a structured argument, while explicitly clarifying the applicable context, the assumptions made, and the justifications given. The role of the assurance case is to make a convincing and valid argument that a system meets its assurance requirements for a given application in a given operating environment.

The work in this paper is motivated, in part, by the following questions: (i) given a tool implementing a formal method, if the tool output will be used as evidence to support claims made in a system/software safety case, why should the output of that tool be trusted? (ii) assuming the output can be considered to be trustworthy, i.e., the tool does, in fact, implement the formal method, is the output suitable to support the claims for which it will be used as evidence?

The first question is an issue of tool assurance; prescriptive certification recognizes the need to assure that tools adequately perform their intended function, and recommends tool *qualification*, e.g., using DO-330 [2], based upon whether they are used during development or verification, and their potential safety impact. The approach for assurance is essentially similar to that in DO-178C, i.e., by appeal to a set of process objectives. The second question is an issue of the appropriateness of applying a formal method, or more specifically, the results of that formal method for software safety assurance. Additional prescriptive guidance exists for applying formal methods, e.g., DO-333 [3]. Again, the approach for assurance is similar to DO-178C, although an additional set of objectives apply.

We make two observations here: first, in the same way as for prescriptive certification of software [4], the link to safety is only implicit through an appeal to the satisfaction of process objectives when we assure (qualify) a formal method (tool) to be used for software safety assurance. That is, the rationale for using product-specific evidence and argumentation to assure system/software safety, is equally applicable for assuring formal methods/tools when they are applied in the context of safety assurance. Second, the output from formal methods/tools has been included in assurance cases largely as *black-box evidence*, i.e., usually only the result of applying the formal method/tool is referenced. In this case, there exists an assurance deficit [5] about the evidence asserted to support a claim, since little can be substantiated about the trustworthiness and suitability of the tool output in the context of the (higher-level) system/software requiring assurance.

We assert that an *evidence argument* including additional information, such as the tool-specific claims made, and the assumptions/reasoning underlying the formal method/tool, provides a richer view of tool-based evidence. This argument can be independently scrutinized to reduce the assurance deficit and, consequently, improve the assurance that can be provided.

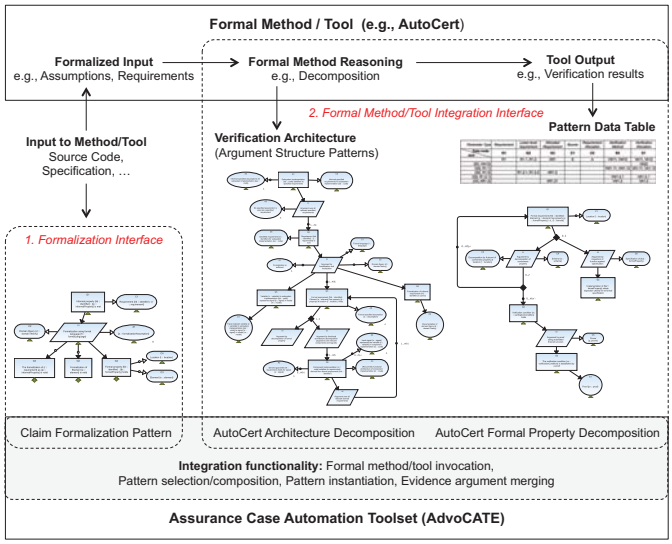

Fig. 1. Tool-based evidence integration architecture (for AUTOCERT)

Previously [6], we showed how such an evidence argument could be created, from the output of a theorem prover based verification tool, and then integrated automatically into the wider assurance case. In this paper, we generalize and improve our previous work. In particular, our paper makes the following contribution: we describe a generic framework for integrating a formal method/tool into an assurance case such that (a) the method/tool can be directly invoked from the environment being used to construct a software assurance case, (b) the output of the tool can be automatically converted into an evidence argument and then integrated into the assurance case from which the tool was invoked.

We illustrate our approach by application to a real example system, the Swift Unmanned Aircraft System (UAS) [7], where we verify a low-level software function in the autopilot, by invoking a code analysis tool from the software safety case, and then transform and integrate the tool output as an evidence argument. We have also implemented this approach in our assurance case toolset, AdvoCATE [8].

II. APPROACH

Fig. 1 shows an architecture for our approach for tool-based evidence integration. Note that although the integration interface shown (dotted box 2, in Fig. 1) is specific to the AUTOCERT code analysis tool [9], the overall architecture is generic and applicable to other formal methods/tools.

The key idea behind our approach is that the integration of a formal method/tool can be specified using *argument structure patterns* (or safety case patterns). Patterns are intended to capture repeatedly used structures of successful (i.e., correct, comprehensive and convincing arguments), within a safety case [10]. In effect, they provide a reusable approach to safety argumentation by serving as a means to capture expertise, known best practices, successful certification approaches, and solutions that have evolved over time.

The existing notion of a pattern is an argument structure, often specified graphically using the goal structuring notation (GSN) [11]. It abstractly captures the reasoning linking certain (types of) claims to the available (types of) evidence, and is accompanied by clear guidelines for its usage. We have extended and formalized patterns to include well-founded recursion and multiplicity constraints [12], which are necessary for capturing the reasoning underlying a tool-based formal verification (e.g., as shown in Fig. 2).

In general, we consider formal verification to have a *divide-and-conquer* form. More specifically, the generic "architecture" of a formal argument is *formalization*, followed by repeated *decomposition* (of various kinds), followed by repeated *solutions* (e.g., function inspection, proof of a verification condition, etc.).Formal verification provides data that we use to instantiate parameters of the patterns (defining the formalization, decompositions, and solutions), thus giving an instance argument. The patterns are also typed, both on the parameters (e.g., function, verification condition) and the argument nodes (e.g., formal claim, assumption, etc.). These types are drawn from a (formal) ontology (not discussed here), which systematically defines the concepts used by a tool.

To integrate a formal method/tool we must provide these patterns and a mapping from tool output to pattern parameters. We specify the mapping as a *pattern data table*, i.e., a tabular form which maps patterns parameters to their possible values drawn from the tool output. Instantiation is algorithmic [12], and it produces a well-formed evidence argument instance that more comprehensively substantiates the tool output.

Effectively, the evidence argument can be considered as a transformation of the existing compendium of information embodied by the formal verification, i.e., formalized claims, specific decompositions and the corresponding formalized subclaims, the relevant assumptions and the low-level solutions, into an argument structure. An advantage of this transformation, we believe, is that tool-based evidence and its suitability for safety assurance is now explicit and can be inspected in a unified way.

III. ILLUSTRATIVE EXAMPLE

To illustrate our approach, we will use AUTOCERT as the target formal tool for integration, and the Swift UAS as the system in which software safety assurance is required. First, we elaborate the formalization and integration interfaces: we briefly describe the nature of verification in AUTOCERT; then, we specify the argument structure patterns embodying verification in AUTOCERT, using GSN. Finally, we exemplify the usage by automatically creating and integrating an evidence argument instance, to support a software safety claim made in a fragment of the autopilot software safety case of the Swift UAS [7].

A. Verification in AUTOCERT

AUTOCERT [9] is a static source-code analysis tool, which performs a two-phase verification: first, given a specified safety policy (e.g., all array indices are within specified bounds) or a

correctness requirement (e.g., code implements a mathematical equation), it uses mathematical annotation schemas to infer logical invariants on the source code; from these, it then generates verification conditions (VCs). In a second phase, VCs are checked using automated theorem provers. The proofs of the VCs constitute the primary verification evidence. However, AUTOCERT also produces an XML output file which encodes the essential information contained in the overall inference: the invariants, the dependencies between the invariants, the verification conditions, axioms used, as well as links to the proofs of the verification conditions. We specify the input to AUTOCERT as a set of assumptions and requirements, each of which is an assertion about the safety/correctness of variables in the code being verified.

The specification formalizes software requirements which, in turn, either derive from system requirements defined during the safety analysis, or are a product of safety analysis applied at the software-level. The formal verification takes place in the context of a logical domain theory, i.e., a set of axioms and function specifications. Axioms can be subjected to increasing levels of scrutiny, going from simply assuming that they are valid, to inspecting them, up to testing them against an executable model which can, itself, be inspected [13].

B. Patterns for Reasoning in AUTOCERT

Prior to invoking a formal method/tool, such as AUTOCERT, informally specified claims must be formalized in the language of the tool. We use the claim formalization pattern (CFP), as shown in Fig. 1, for this purpose. Due to space constraints, we do not discuss this pattern here; however, the structure of the pattern is simply that an informal claim is developed into (one or more) formal sub-claims by using an appropriate logic or formal language. In our case, the formal language is the input language of AUTOCERT.

As discussed in Section II, the next step to integrate a formal verification tool is to capture the reasoning embodied by the tool as an argument structure pattern. For integrating AUTOCERT, two patterns reflect its verification architecture: the AUTOCERT functional architecture decomposition pattern, and the AUTOCERT formal property decomposition pattern.

1) AUTOCERT *Functional Architecture Decomposition:* We only briefly describe this pattern and do not provide its structural specification since, primarily, functional decomposition in AUTOCERT is not invoked for the example claim being verified later (See Section III-C); and secondarily, due to space constraints. The main intent of the AUTOCERT functional architecture decomposition pattern is to encode how the claim that an implementation meets its requirements is developed by logical decomposition over its constituent components, i.e., its *functional architecture*.

The first strategy is to address *localized* requirements relevant to the implementation. A localization claim/requirement may take the form that a variable in the code being verified traces to a valid requirements concept, such as a signal. The sub-claims produced address, respectively, the set of relevant requirements to be demonstrated and their correct *localization*.

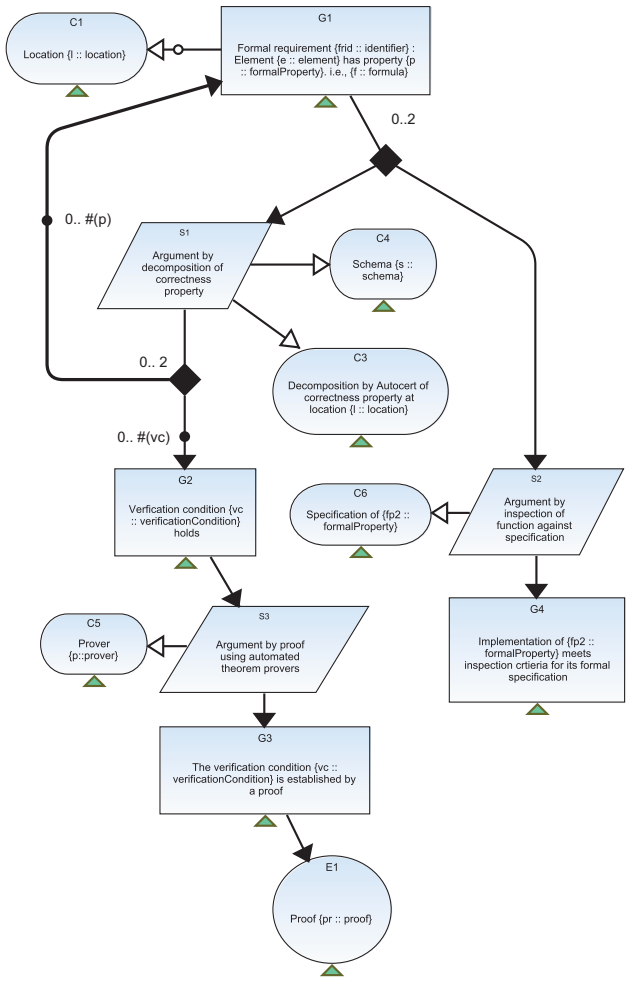

Fig. 2. AUTOCERT formal property decomposition pattern

AUTOCERT determines the traces for correct localization, and is represented in the pattern as a parameteric solution node.

Then, to establish that the requirements relevant for the implementation are met, as part of its analysis AUTOCERT derives the functional architecture of the implementation. We represent this as the strategy that the constituent component-level requirements are met, in the context of the relevant components and signals of the recovered functional architecture. In this step, AUTOCERT derives guarantees about certain connected components, which are used as assumptions in other components, and also the localization of the relevant variables in the code. Since each formal component may itself have multiple derived requirements, and since the functional architecture can be hierarchically organized, the procedure is iterative. We represent iteration in the pattern using the loop construct [12]. When an *atomic* requirement is reached, i.e., it is the lowest-level requirement, then AUTOCERT applies property decomposition, which we describe next.

2) AUTOCERT *Formal Property Decomposition:* Fig. 2 shows the AUTOCERT formal property decomposition pattern, whose intent is to substantiate a formal requirement by formal

property decomposition, and/or inspection.

The top-level claim, G1, of this pattern, is that a formal requirement, i.e., a *property* of an *element*, holds at a specific *location*. An element can be code, a formal model, etc., whereas a location can be a specific line number for code, a file, etc. We develop this top-level claim using one of two strategies: a formal decomposition strategy, S1; or inspection, S2. Applying S1 produces either one or more sub-claims of the same type as the parent claim, or one or more VCs, G2. The former represents an iterative decomposition (reflected by the loop from S1 to G1). To develop the latter, AUTOCERT calls an appropriate theorem prover (Context C5) and the underlying strategy (S3) is a proof of correctness. The resultant sub-claim (G3) is that proof exists (solution E1) for the VC. Note, here, that S2 represents the condition when AUTOCERT fails to decompose a formal requirement or produce a VC, therefore requiring human interaction for continuing verification.

C. Usage for Software Assurance

To illustrate the usage of our approach, we verify a low level software claim for the Swift UAS autopilot: that the implementation of the controller computing the value of the aileron control variable, during descent, is correct[1].

Fig. 3 shows the fragment of the autopilot software safety case where this claim is made (the argument structure connecting the nodes G4 through G7, where the goal node G7 is the claim of interest). To develop G7, first we formalize it by applying the CFP. In our implementation in AdvoCATE, either the end-user interactively supplies the CFP parameter values, or they can be drawn from an AUTOCERT *certification file*. Thereafter, our instantiation algorithm [12] automatically creates the instance (shown in Fig. 3 as the argument structure connecting the goal nodes G7–G9 and G53).

The claim in goal node G53 is the formalized equivalent (using the AUTOCERT input language) of the informal requirement in goal node G7. This is the point at which we invoke AUTOCERT for verification. Based on the source code under verification, AUTOCERT can determine whether to apply functional architecture decomposition before property decomposition. In this case, only the property decomposition applies. Subsequently, we map the verification output produced to a pattern data table consistent with the AUTOCERT formal property decomposition pattern (Fig. 2), which is then used for instantiation.

The instance argument structure produced, i.e., the evidence argument, encodes the reasoning and the output from AUTOCERT formal verification, which is then grafted onto the node from which the tool was invoked. In Fig. 3, this is the argument structure from node G53 (inclusive) downwards. Note that developing claim G53 into sub-claim G5 employs strategy S33, which is identical to the child strategy, S5, of the sub-claim G5. This corresponds to one unrolling of the loop in the AUTOCERT formal property decomposition pattern, i.e., what is shown is only one step in the verification.The AUTOCERT output determines the number of loop unrollings, which the instantiation algorithm uses to create the complete evidence argument, shown as a bird's eye view in Fig. 4.

[1]Note that although this is a correctness requirement, it is also a safety requirement as determined through safety analysis of the Swift UAS and its software system.

IV. DISCUSSION

Currently, our implementation in AdvoCATE has a hard-coded integration of AUTOCERT. However, thus far, we have only described one aspect of the integration, i.e., the specification of the patterns representing the reasoning underlying a formal method/tool. To make the infrastructure fully extensible, we need to consider the data that should be associated with a tool and how it should be specified. We specify the tool integration as embeddings into the three pattern types: formalization, decomposition, and solution (i.e., divide and conquer). Then, we need to specify:

(i) the set of patterns giving the architecture of the verification: namely formalization, decomposition, and solutions, i.e., effectively, the *semantics* of the tool. Depending on the formal method/tool, there can be multiple forms of decomposition and solution.

(ii) a mapping from tool output to pattern parameters: the mapping could be expressed in an instantiation file (i.e., a file where we specify a mapping between XML tags, say, and pattern parameters).

(iii) the *syntax* of the tool input and output: the input should be specified in terms of an ontology. The tool input syntax relates to the formalization pattern.

(iv) how to invoke the tool, i.e., commands and arguments.

Effectively, by encoding the reasoning underlying a formal method as a pattern that can be independently inspected, we specify a basis to gauge the trustworthiness of that method. Furthermore, by instantiating the pattern, we create concrete links from the verification items, embodying tool-based evidence, to the relevant system/software safety claims. We believe this offers an approach to judge the suitability and integrity of the tool-based evidence asserted to support the system/software safety claims.

An advantage of an evidence argument for a formal method/tool is that it allows formal reasoning and evidence to be integrated into the language of assurance arguments rather than be treated as separate artifacts. This provides a simple form of *evidence management* [14] and the ability to drill down to details of a verification, from within the argument itself. It also allows higher-level requirements and the corresponding claims, to be (automatically) traceable to low-level evidence.

Hierarchical abstraction [18] of the evidence argument (not shown in this paper), allows us to hide some proof details and only expose in the argument, those steps that are relevant for tracing the formalized requirements to both code and higher-level requirements.

Assurance cases have been shown to be both complementary to, and compatible with, prescriptive safety certification [4]; we assert that the same is true also of evidence arguments. To illustrate, we briefly describe how an evidence argument can

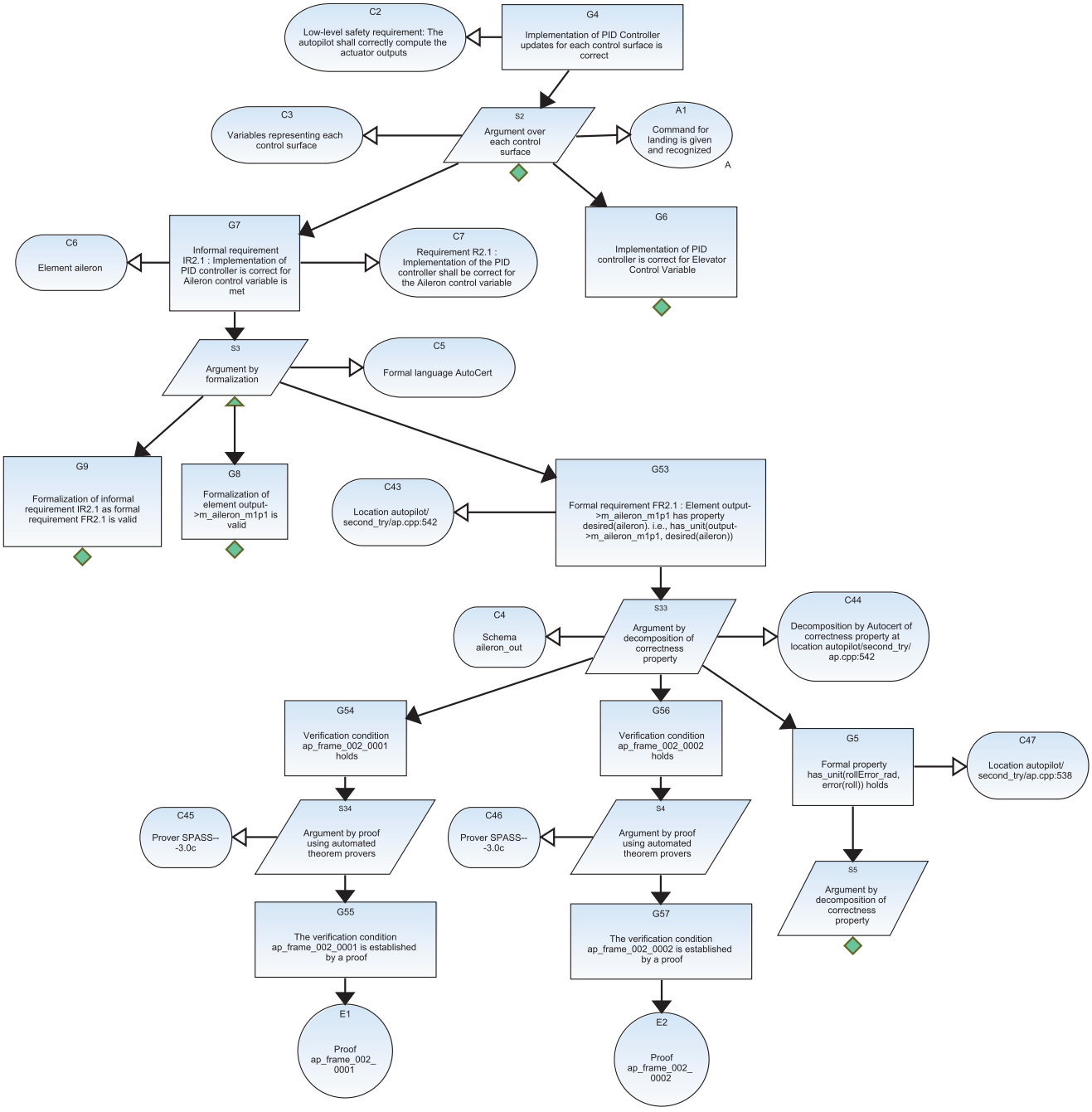

Fig. 3. Fragment of the autopilot software safety case, containing a formalized claim, substantiated by an evidence argument generated from the output of AUTOCERT and the integration interface. Only one step in the evidence argument has been shown.

be mapped to some of the objectives in DO-333 [3]: the prescriptive guidance for using formal methods with DO-178C [1]. For example, each claim corresponding to a requirement (e.g., the claim in goal node G7 in Fig. 4) that is formalized using the CFP (e.g., the claim in goal node G33 in Fig. 4), the evidence argument explicitly links to the solutions that justify them (e.g., the proof references in solution nodes E1, E2, and so on). These traces map to objectives FM3 and FM4 (coverage of high-level, and low-level requirements) in the Table FM.C7 of DO-333: *Verification of Verification Process Objectives*. Furthermore, the evidence argument explicitly links code to the relevant requirements (e.g., the link from context nodes C43, C44, C47, upwards to goal node G7, via goal node G53), which maps to objective 5 (source code traceability to low-level requirements) in Table FM.C5 of DO-333: *Verification of Outputs of Software Coding and Integration Process*. Other such mappings also exist between the evidence argument and many of the objectives in DO-333.

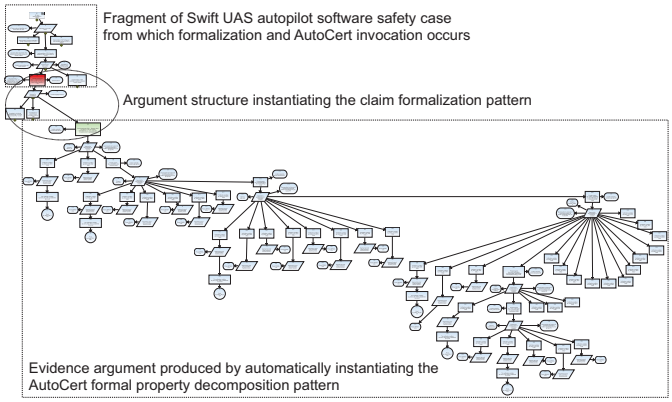

Fig. 4. Bird's eye view of the integrated evidence argument: AUTOCERT invoked on the formal claim produces verification data, used to instantiate the AUTOCERT formal property decomposition pattern; then the instance is grafted onto the original safety case fragment.

This paper generalizes our earlier ideas on integrating formal methods into assurance cases [6], to enable the principled integration of other formal methods/tools. Other integration frameworks also exist [15] for producing/maintaining claims supported by formally generated evidence. The distinction from our work is the focus on integrating different tools from the viewpoint of capturing a distributed workflow. Other workflow-based methods for integrating evidence from formal verification methods have been considered in [16]; however, evidence is included as a reference to verification outputs as in current practice, in contrast with our new approach for integrating the entire evidence argument. *Confidence arguments* [5] are a complementary approach to ours for reducing assurance deficits by specifying the sources of assurance deficits in evidence, and justifying their mitigation; whereas an evidence argument highlights the reasoning underlying specific formal method/tool output that is used as evidence. Our work is also closely related to the *evidence metamodel* [17] which describes the interface between arguments and evidence.

V. CONCLUSIONS AND OUTLOOK

To address the questions of suitability and trustworthiness of tool-based evidence included in a software assurance case (Section I), our solution is to include an evidence argument instead of, simply, a reference to the output of the tool.

Our tool, AdvoCATE, provides a principled basis for formalizing claims, e.g., using the CFP. Together with our architecture for tool-based evidence integration we can support formalized claims through evidence arguments, generated by instantiating patterns, encoding formal verification rationale, using the verification output from a corresponding formal tool. We believe this architecture is generic enough also to be able to integrate other formal methods/tools, including those for deductive verification, abstract interpretation and model checking. In fact, AUTOCERT, is an example of a formal tool based upon deductive verification; we also have a preliminary integration interface for the tool, IKOS (*Inference Kernel for Open Static Analyzers*), whose underlying formal method is abstract interpretation.

An advantage of our implementation, is that AdvoCATE can provide a unified and convenient interface for using formal verification tools. Currently, it only allows invoking the tool (and integrating the results into arguments), but we envision unified mechanisms for reporting results as well as choosing tools. However, our approach has not yet addressed the argument about the tool itself, i.e., its *qualification*, or the *extent of assurance* that a particular formal method/tool can provide. We believe that these are potentially promising areas for future work.

ACKNOWLEDGMENT

This work was funded by the AFCS element of the SSAT project in the Aviation Safety Program of NASA ARMD.

REFERENCES

[1] RTCA Inc., SC-205, "Software Considerations in Airborne Systems and Equipment Certification," DO-178C, Dec. 2011.
[2] RTCA Inc., SC-205, "Software Tool Qualification Considerations," RTCA DO-330, Dec. 2011.
[3] RTCA Inc., SC-205, "Formal Methods Supplement to DO-178C and DO-278A," RTCA DO-333, Dec. 2011.
[4] R. Hawkins, I. Habli, T. Kelly, and J. McDermid, "Assurance cases and prescriptive software safety certification: A comparative study," *Safety Science*, vol. 59, pp. 55 – 71, 2013.
[5] R. Hawkins, T. Kelly, J. Knight, and P. Graydon, "A new approach to creating clear safety arguments," in *Proc. Safety Critical Systems Symp.*, Feb. 2011.
[6] E. Denney, G. Pai, and J. Pohl, "Heterogeneous aviation safety cases: Integrating the formal and the non-formal," in *17th IEEE ICECCS 2012*, Jul. 2012, pp. 199–208.
[7] E. Denney, G. Pai, and J. Pohl, "Automating the generation of heterogeneous aviation safety cases," NASA Ames Research Center, Technical Report NASA/CR-2011-215983, Aug. 2011.
[8] E. Denney, G. Pai, and J. Pohl, "AdvoCATE: An Assurance Case Automation Toolset," in *SAFECOMP 2012 Workshops*, LNCS vol. 7613, 2012, pp. 8–21.
[9] E. Denney and S. Trac, "A software safety certification tool for automatically generated guidance, navigation and control code," in *IEEE Aerospace Conf. Electronic Proc.*, 2008.
[10] T. Kelly and J. McDermid, "Safety case construction and reuse using patterns," in *SafeComp '97*, 1997, pp. 55–69.
[11] Goal Structuring Notation Working Group, "GSN Community Standard Version 1," Nov. 2011. [Online]. Available: http://www.goalstructuringnotation.info/
[12] E. Denney and G. Pai, "A Formal Basis for Safety Case Patterns," in *SAFECOMP 2013*, LNCS, vol. 8153, 2013.
[13] K. Y. Ahn and E. Denney, "A framework for testing first-order logic axioms in program verification," *Software Quality Journal*, pp. 1–42, Nov. 2011.
[14] Object Management Group, "Structured Assurance Case Metamodel (SACM) version 1.0," Formal/2013-02-01, Feb. 2013.
[15] S. Cruanes, G. Hamon, S. Owre, and N. Shankar, "Tool integration with the evidential tool bus," in *VMCAI 2013*, LNCS, vol. 7737, 2013, pp. 275–294.
[16] G. Juhász, "Verification processes in certification of safety critical systems," M.S. thesis, Budapest University of Technology and Economics, 2011.
[17] L. Sun, "Establishing confidence in safety assessment evidence," Ph.D. dissertation, University of York, 2012.
[18] E. Denney, G. Pai, and I. Whiteside, "Hierarchical safety cases," in *NFM 2013*, LNCS, vol. 7871, May 2013, pp. 478–483.