2009 IEEE/ACM International Conference on Automated Software Engineering

A Verification-Driven Approach to Traceability and Documentation for Auto-Generated Mathematical Software

Ewen Denney
SGT / NASA Ames
Moffett Field, CA 94035
Ewen.W.Denney@nasa.gov

Bernd Fischer
School of Electronics and Computer Science
University of Southampton, England
B.Fischer@ecs.soton.ac.uk

***Abstract*—Automated code generators are increasingly used in safety-critical applications, but since they are typically not qualified, the generated code must still be fully tested, reviewed, and certified. For mathematical and engineering software this requires reviewers to trace subtle details of textbook formulas and algorithms to the code, and to match requirements (e.g., physical units or coordinate frames) not represented explicitly in models or code. We support these tasks by using the AutoCert verification system to identify and verify mathematical concepts in the code, recovering verified traceability links between concepts, code, and verification conditions. We then exploit these links to construct a natural language report that provides a high-level structured argument explaining where the code uses specified assumptions and why and how it complies with the requirements. We have applied our approach to generate review documents for several subsystems of NASA's Project Constellation.**

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 [10], there is no guarantee that their output is correct, and the generated code must thus still be fully tested, reviewed and certified. This requires reviewers to match subtle details of textbook formulas and algorithms to the code, which is often difficult to understand. Moreover, common modeling languages do not allow important domain requirements to be represented explicitly (e.g., coordinate frames); consequently, the generated code is not traced back to them.

The central problem is to disentangle the complexity of the generated code, in order to build up a comprehensible explanation in terms of high-level domain concepts. This in turn requires comprehensive traceability links that associate the code not only with the model and verification artifacts, but also with abstract concepts and requirements such as coordinate frames. The challenge is to recover these traceability links: we cannot assume that the code generator will provide them nor can we rely on the correctness of any links that are provided. In fact, we need explicit assurance that the traceability links are correct, because any documentation derived from them could be misleading otherwise.

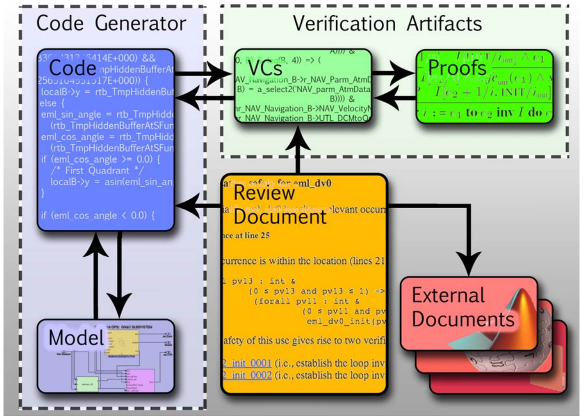


Figure 1. Tracing between artifacts

Here we describe a verification-driven approach to traceability and documentation. Its central insight is that we can combine methods from program understanding and program verification to recover *verified traceability links* from which we can construct natural language documentation that explains why and how automatically generated code complies with specified requirements. The documentation lists the external assumptions on the code (e.g., the physical units and constraints on input signals), the dependencies between variables, and the algorithms, data structures, and conventions (e.g., quaternion handedness) used by the generator to implement the model. It also shows how assumptions and requirements are related through the code, in particular, the complete chain of reasoning which allows the requirements to be concluded from the assumptions, which assumptions are used to show a specific requirement, and which assumptions remain unused. The documentation is hyper-linked to both the program and the verification conditions, and so gives traceability between verification artifacts, documentation, and code (see Figure 1).

The approach described here is based on the AutoCert code analysis tool [2], which takes a set of requirements, and formally verifies that the code satisfies them. AutoCert can verify execution-safety requirements (e.g., array

1527-1366/09 $29.00 © 2009 IEEE
DOI 10.1109/ASE.2009.71

bounds), as well as domain-specific requirements such as frame safety. Our approach is generator-independent, and we have used it with several different in-house and commercial code generators, including Real-Time Workshop. In particular, we have applied our tool to several subsystems of the GN&C software currently under development for the Constellation program, and used it to generate review reports for domain-specific requirements such as the consistent use of Euler angle sequences and coordinate frames.

II. Software Certification using AutoCert

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. AutoCert follows the Hoare logic approach to verification; hence, it needs *annotations*, i.e., logical assertions of program properties, at key locations in the code. These annotations are constructed automatically by a post-generation inference phase that exploits the idiomatic nature of auto-generated code and is driven by a generator- and domain-specific set of idioms. The inference algorithm builds an abstracted control-flow graph (CFG), collapsing the code idioms into single nodes. It then traverses the CFG from *use* nodes (where a requirement must be shown) backwards to all corresponding *definitions* (where the relevant properties are ultimately established) and annotates the statements along the paths as required [2]. A verification condition generator (VCG) processes the annotated code, feeding a set of verification conditions (VCs) into an automated theorem prover (ATP); their proofs guarantee that the code satisfies the requirements.

AutoCert is implemented as a generator-independent plug-in: since it only analyzes the code and not the model or the generation process, the generator is treated as a black box. The inference is customized via a set of high-level *annotation schemas* [4], which use patterns to describe code idioms and actions to construct the annotations needed to certify a matching code fragment. The schemas remain untrusted, since the assurance does not rely on their correctness, but follows from the proofs of the corresponding VCs. The schemas also contain textual descriptions which can be parametrized by the variables in the pattern, and slots for recording other information associated with the code, such as the mathematical conventions it uses.

An annotation schema compiler [4] takes a set of schemas, and compiles them down into customized annotation and documentation templates, and concept relations drawn from a domain ontology. The annotation templates are then used to produce the annotated program, while the other elements are used during the document generation process. Hence, schemas are central to achieving our goal of a unified approach to verification, documentation, and tracing.

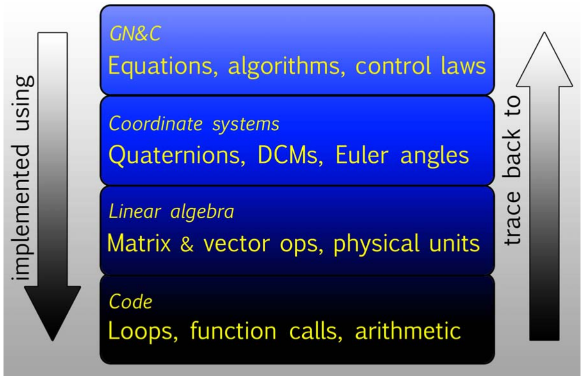

implemented using
GN&C
Equations, algorithms, control laws
Coordinate systems
Quaternions, DCMs, Euler angles
Linear algebra
Matrix & vector ops, physical units
Code
Loops, function calls, arithmetic
trace back to

Figure 2. Levels of domain abstraction

III. Verified Traceability Link Recovery

Link Categories. Traceability links associate different entities from different development phases with each other. In model-based development, they are generally equated with links between the individual elements of the model (e.g., Simulink boxes) and their representation in the generated code. Most commercial code generators add them directly to the generated code (e.g., as comments or embedded hyperlinks), and academic research has worked on maintaining and recovering these links after model or code changes [1].

However, this view is too restrictive for our purposes. First, the certification process is driven by a set of mission-specific requirements, and the documents must be structured according to these; consequently, the traceability links must go back to these requirements as well. Second, the documents need to explain the code in terms of high-level domain concepts not explicitly represented by model elements. Figure 2 shows some concepts of the GN&C domain at different levels of abstraction. At the lowest level, at which a code review is actually carried out, is the code itself along with primitive arithmetic operators. At the next level are mathematical operations, such as matrix multiplication and transpose, and physical values of a given unit, corresponding to the low-level datatypes. These, in turn, are used to represent navigational information in terms of quaternions, directional cosine matrices (DCMs), Euler angles, etc., in various coordinate systems. This is the level at which we explain the verification. At the highest level of abstraction are the GN&C principles, themselves, but explanation at this level is currently beyond our scope. Third, the documents also need to explain the *internal structure* of the code; in particular, we need to recover links reflecting the chains of implications from the properties of one variable to the properties of one or more dependent variables in order to show how a requirement ultimately follows through the code from the assumptions. We thus distinguish several link categories, depending on the entities related to each other.

Requirement-to-concept links relate the individual requirements to the concepts in the upper tiers of the domain abstractions (see Figure 2). They represent the set of data structures, conventions, and operations used to implement a given requirement, which is the most important information from an understanding point of view. *Requirement-to-code* links trace individual requirements back to the lines of code implementing them. They can be used to show that the implementation does not contain any superfluous functionality.

Requirement-to-assumption links make explicit on which of the specified assumptions the validity of a requirement rest. This is the most important information from a certification point of view. Similarly, *requirement-to-axiom* links relate requirements to the specific domain theory axioms that are used by the ATP to prove the associated VCs. Note that most ATPs treat assumptions and axioms interchangeably, but because they play different roles in the certification, we need two different link categories: the assumptions are specific to the code, and need to be established by other system components, while the axioms represent the logical formulation of domain concepts and are "hardened" over time and are thus more trusted. *Requirement-to-VC* links primarily serve book-keeping (rather than understanding) purposes and show which requirements are "at risk" if VCs have not been proven yet. They are also used to compute the links in the two categories above, since the VCs give access to the proofs.

Code-to-code links reflect the internal conceptual structure of the code, not its syntax: two code locations are linked if they are directly connected by an edge in the abstracted CFG built by the annotation inference. Simple *code-to-VC* links are provided by most formal verification tools but links based on a categorization of the VCs according to their purpose (e.g., establishing a definition or showing the safety of a use location) allow a more fine-grained linking. These links can also pinpoint the location of faults, if a VC fails.

Link Recovery via Annotation Inference. The core traceability links required to generate meaningful documentation are code-to-code and requirement-to-concept links. The former are recovered by AUTOCERT's annotation inference, as side effect of the CFG traversal. Recovery of the latter is based on the fact that matching a schema against the code is actually performing program understanding, and that the schema itself thus already reflects all domain concepts that can be extracted from the matched code fragment—in effect, the schema already tells us everything we need to know about the code. Since the match is verified if all associated VCs are proven, the requirement-to-concept traceability links are verified as well.

Consider for example the *DCM-NED-to-Nav* schema shown in Figure 3, which looks for a sequence of assignments with values corresponding to the DCM's entries. If it is matched against some code, and all VCs are proven, then we know for certain that the code is related to the *DCM* from *NED* to *Nav* concept and, in particular, represents a DCM as a 9-vector, as stated by the schema's concept list. Hence, we have recovered two verified code-to-concept links, and since we know the requirement currently being certified, also two requirement-to-concept links.

```
schema(dcm_ned_nav=['a DCM from NED to Nav']
, frame
, def(V)
, ((V[0] := x0) : : (x0 ~= cos(H − A);
   (V[1] := x1) : : (x1 ~= -sin(H − A);
   (V[2] := x2) : : (x2 ~= 0);
   (V[3] := x3) : : (x3 ~=  sin(H − A);
   (V[4] := x4) : : (x4 ~=  cos(H − A);
   (V[5] := x5) : : (x5 ~= 0);
   (V[6] := x6) : : (x6 ~= 0);
   (V[7] := x7) : : (x7 ~= 0);
   (V[8] := x8) : : (x8 ~= 1)
  ) ← &(post frame(V, dcm(ned, nav)),
        pre ∃ ψ, φ · unit(ψ, heading) ∧ unit(φ, azimuth)
            ∧ x0=cos(ψ − φ) ∧ x1=-sin(ψ − φ) ∧ x2=0
            ∧ x3=sin(ψ − φ) ∧ x4=cos(ψ − φ) ∧ x5=0
            ∧ x6=0 ∧ x7=0 ∧ x8=1),
, [dcmrep=vec(9)]
).
```

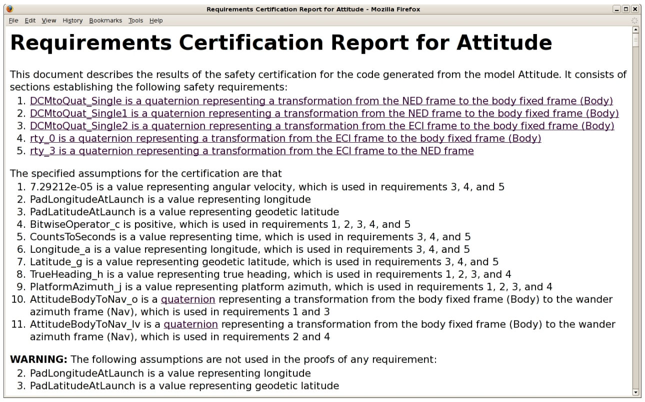
Figure 3. Annotation schema

Link Recovery via Verification. The VCG is primarily responsible for the code-to-VC links. It adds the relevant source code locations to the generated VCs; however, these need to be maintained by subsequent processing steps, e.g., simplification. Here, we use our previous work on semantic labeling [3] to achieve the VC categorization and fine-grained linking. The requirement-to-assumption and requirement-to-axiom links are extracted from the *proofs* of the VCs. This is in principle a simple task, but it requires the ATP to provide an explicit proof output. We currently only analyze proofs in the standard TPTP proof notation [11].

IV. GENERATING REVIEW DOCUMENTS

Document Structure. The generated documents are intended as structured reading guides for the code and the verification artifacts, showing why and how the code complies with the specified requirements. The introduction contains a natural language representation of the formalized requirements and certification assumptions; see Figure 4 for an example. This allows the reviewers to check that their formalization has not (inadvertently) introduced any conceptual mismatches. It also contains the requirement-to-assumption links recovered by the proof analysis, and calls out assumptions that are not necessary.

Each requirement section starts with a summary of the the relevant variables and the high-level conventions and operations used by the code, and concludes with a series of subsections that explain why and how each of the variables


Requirements Certification Report for Attitude - Mozilla Firefox
File Edit View History Bookmarks Tools Help

Requirements Certification Report for Attitude

This document describes the results of the safety certification for the code generated from the model Attitude. It consists of sections establishing the following safety requirements:

1. DCMtoQuat_Single is a quaternion representing a transformation from the NED frame to the body fixed frame (Body)
2. DCMtoQuat_Single1 is a quaternion representing a transformation from the NED frame to the body fixed frame (Body)
3. DCMtoQuat_Single2 is a quaternion representing a transformation from the ECI frame to the body fixed frame (Body)
4. rty_0 is a quaternion representing a transformation from the ECI frame to the body fixed frame (Body)
5. rty_3 is a quaternion representing a transformation from the ECI frame to the NED frame

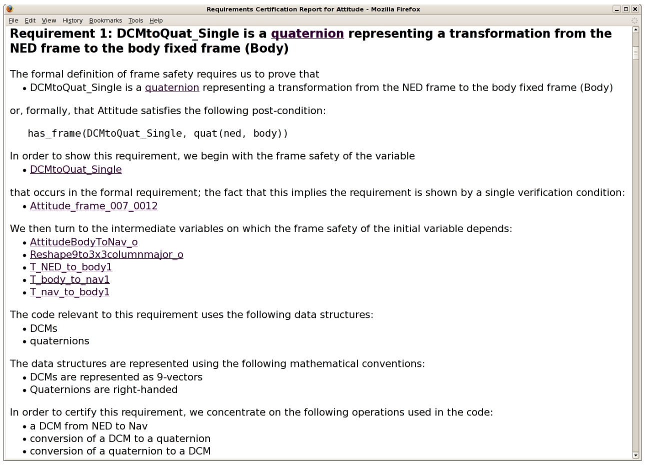
The specified assumptions for the certification are that

1. 7.29212e-05 is a value representing angular velocity, which is used in requirements 3, 4, and 5
2. PadLongitudeAtLaunch is a value representing longitude
3. PadLatitudeAtLaunch is a value representing geodetic latitude
4. BitwiseOperator_c is positive, which is used in requirements 1, 2, 3, 4, and 5
5. CountsToSeconds is a value representing time, which is used in requirements 3, 4, and 5
6. Longitude_a is a value representing longitude, which is used in requirements 3, 4, and 5
7. Latitude_g is a value representing geodetic latitude, which is used in requirements 3, 4, and 5
8. TrueHeading_h is a value representing true heading, which is used in requirements 1, 2, 3, and 4
9. PlatformAzimuth_j is a value representing platform azimuth, which is used in requirements 1, 2, 3, and 4
10. AttitudeBodyToNav_o is a quaternion representing a transformation from the body fixed frame (Body) to the wander azimuth frame (Nav), which is used in requirements 1 and 3
11. AttitudeBodyToNav_lv is a quaternion representing a transformation from the body fixed frame (Body) to the wander azimuth frame (Nav), which is used in requirements 2 and 4

WARNING: The following assumptions are not used in the proofs of any requirement:

2. PadLongitudeAtLaunch is a value representing longitude
3. PadLatitudeAtLaunch is a value representing geodetic latitude

Figure 4. List of requirements and assumptions


Requirements Certification Report for Attitude - Mozilla Firefox
File Edit View History Bookmarks Tools Help

Requirement 1: DCMtoQuat_Single is a quaternion representing a transformation from the NED frame to the body fixed frame (Body)

The formal definition of frame safety requires us to prove that

- DCMtoQuat_Single is a quaternion representing a transformation from the NED frame to the body fixed frame (Body)

or, formally, that Attitude satisfies the following post-condition:

```
has_frame(DCMtoQuat_Single, quat(ned, body))
```

In order to show this requirement, we begin with the frame safety of the variable

- DCMtoQuat_Single

that occurs in the formal requirement; the fact that this implies the requirement is shown by a single verification condition:

- Attitude_frame_007_0012

We then turn to the intermediate variables on which the frame safety of the initial variable depends:

- AttitudeBodyToNav_o
- Reshape9to3x3columnmajor_o
- T_NED_to_body1
- T_body_to_nav1
- T_nav_to_body1

The code relevant to this requirement uses the following data structures:

- DCMs
- quaternions

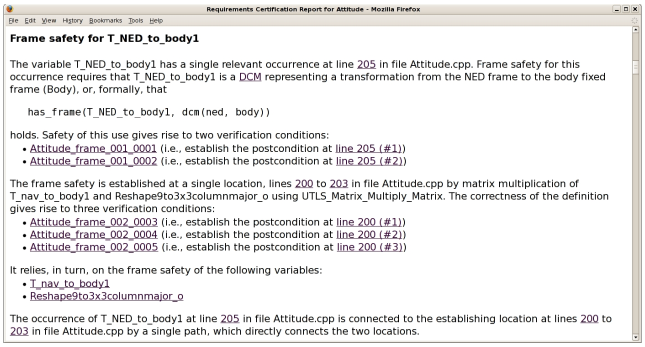
The data structures are represented using the following mathematical conventions:

- DCMs are represented as 9-vectors
- Quaternions are right-handed

In order to certify this requirement, we concentrate on the following operations used in the code:

- a DCM from NED to Nav
- conversion of a DCM to a quaternion
- conversion of a quaternion to a DCM

Figure 5. Operations and conventions


Requirements Certification Report for Attitude - Mozilla Firefox
File Edit View History Bookmarks Tools Help

Frame safety for T_NED_to_body1

The variable T_NED_to_body1 has a single relevant occurrence at line 205 in file Attitude.cpp. Frame safety for this occurrence requires that T_NED_to_body1 is a DCM representing a transformation from the NED frame to the body fixed frame (Body), or, formally, that

```
has_frame(T_NED_to_body1, dcm(ned, body))
```

holds. Safety of this use gives rise to two verification conditions:

- Attitude_frame_001_0001 (i.e., establish the postcondition at line 205 (#1))
- Attitude_frame_001_0002 (i.e., establish the postcondition at line 205 (#2))

The frame safety is established at a single location, lines 200 to 203 in file Attitude.cpp by matrix multiplication of T_nav_to_body1 and Reshape9to3x3columnmajor_o using UTLS_Matrix_Multiply_Matrix. The correctness of the definition gives rise to three verification conditions:

- Attitude_frame_002_0003 (i.e., establish the postcondition at line 200 (#1))
- Attitude_frame_002_0004 (i.e., establish the postcondition at line 200 (#2))
- Attitude_frame_002_0005 (i.e., establish the postcondition at line 200 (#3))

It relies, in turn, on the frame safety of the following variables:

- T_nav_to_body1
- Reshape9to3x3columnmajor_o

The occurrence of T_NED_to_body1 at line 205 in file Attitude.cpp is connected to the establishing location at lines 200 to 203 in file Attitude.cpp by a single path, which directly connects the two locations.

Figure 6. Correctness justification

contributes to the requirement. The subsections can contain explanations of fragments of code, and can refer to the explanations for other variables, which are cross-linked. The document's overall structure thus reflects the way the annotation inference has analyzed the program, starting with the variables occurring in the original requirements.

Whenever AUTOCERT has carried out some analysis using the prover (e.g., that a code fragment establishes some property), the document provides links to the corresponding VCs. A semantic labeling of the VCs [3] allows us to associate only the small number of VCs with the code fragment that actually contribute to demonstrating how a given requirement holds for the fragment. This provides a "natural" high-level grouping mechanism for the VCs, which helps reviewers to focus their attention on the artifacts and locations that are relevant for each requirement.

Explaining Inferred Operations and Conventions. As a result of its analysis, AUTOCERT effectively "reverse engineers" the code, and, based on the information specified in the schemas, identifies both the high-level mathematical structures that are used by the operations relevant to the current requirement, e.g., DCMs and quaternions, and the lower-level data structures used to represent these, e.g., matrices and vectors, including any underlying conventions that manifest themselves in the lower-level data structures (e.g., quaternion handedness). This analysis also identifies cases where several lower-level data structures are used to represent a high-level concept, such as three 3-vectors representing a DCM.

The report contains a concise summary of this information, going from the abstract mathematical structures to the the concrete operations; see Figure 5 for an example. In each category, the entries are grouped by sub-categories, so that for example all extracted information concerning the representation of DCMs is together. This sub-categorization is derived from an underlying concept ontology and the concept lists of the applied schemas. It highlights potential problems caused by different representations used in different parts of the model or by different operations (e.g., the representation of DCMs as 9-vectors and three 3-vectors), and directs the reviewers' attention to this for further inspection and clarification.

Explaining Correctness. The backbone of the document is a chain of implications from the properties of one variable to the properties of one or more dependent variables, corresponding to the recovered code-to-code links. The chain starts at those key variables which appear in the requirement, and continues to variables in the assumptions or input signals. Figure 6 shows one step in this chain.

At this step in the justification, we need to show that the variable `T_NED_to_body1` is a DCM from NED to the Body frame. First, we show that the information which has been inferred at this point in the code does indeed give the variable the required properties, themselves expressed as a post-condition. Two VCs establish this (cf. "safety of this use"). Second, the location where the variable is defined is given, and the correctness of that definition is established, i.e., that it does define the relevant form of DCM, which gives rise to three VCs. Third, we observe that this definition—a matrix multiplication—depends, in turn, on properties of other variables, i.e., the multiplicands, with which the explanation continues later in the document. Fourth, we show that the properties of the definition are sufficient to imply the properties of the use, and that these properties are preserved along the path connecting the two locations; however, in this example, the path is straightforward and does not induce any further VCs.

Summarizing Proofs. Proofs found by ATPs are typically very big, even for simple conjectures. It is thus necessary to summarize the pertinent information, instead of verbalizing the proofs themselves, as for example done in [5]. First, we group axioms into theories and, in some cases, only list the theory instead of the individual axioms. For example all arithmetic reasoning is hidden under a single entry. More relevant axioms representing frame reasoning are listed individually. Since the axiom names are internal and convey no meaning to a reviewer, we associate explanatory texts with the categories. Second, we combine information from entire VC sets, again using recovered traceability links to identify conceptually related VCs.

V. Conclusion

Program comprehension and program verification are fundamentally related activities, because they both need to understand concepts that are distributed throughout the code. Early program comprehension techniques such as *plans* [8] or *focusing* [9] are similar to weak forms of our schemas. Techniques based on approximations to structural and behavioral patterns have also been used by Antkiewicz et al. [1] to reverse engineer framework-specific models from framework code. Similarly to our work these techniques also use patterns to reverse engineer "logical structure", but they do not verify the selected fragments. The Whyline tool [6], which uses techniques from program analysis to answer user queries about program behavior, is closer to our approach in that verification techniques are used to provide explanations in terms of concepts from a domain. However, none of the existing approaches can provide verified traceability links between domain concepts, requirements, code fragments, and verification artifacts, or can construct a high-level natural language explanation for why the specified requirements follow from the assumptions and a background domain theory. AutoCert's documentation feature is aimed at facilitating code reviews for auto-generated code, thus increasing trust in otherwise opaque code generators without excessive manual V&V effort, and better enabling the use of automated code generation in safety-critical contexts.

We are currently working to automate linking of inferred concepts to a mission ontology database, which has been mandated by NASA's Constellation program. The idea is that by automatically annotating the code with inferred concepts, engineers are relieved of this documentation chore. We also plan to provide links to mission requirements documents and other relevant project documentation. Further scaling will require better hierarchy and abstraction mechanisms, and more top-level summaries. Listing formulas and equations that are used in the code would also be helpful for reviews, since ultimately these need to be scrutinized by domain experts. More ambitiously, we seek to further raise the level of abstraction at which the code is explained to the algorithmic level. The ideas of Koellman and Goedicke [7] on recognizing algorithms might prove useful there.

References

[1] M. Antkiewicz, T. Tonelli Bartolomei, and K. Czarnecki. Automatic extraction of framework-specific models from framework-based application code. In *ASE'07*, pp. 214–223. ACM, 2007.

[2] E. Denney and B. Fischer. A generic annotation inference algorithm for the safety certification of automatically generated code. In *GPCE'06*, pp. 121–130. ACM, 2006.

[3] E. Denney and B. Fischer. Explaining verification conditions. In *AMAST'08*, *LNCS 5140*, pp. 145–159. Springer, 2008.

[4] E. Denney and B. Fischer. Generating customized verifiers for automatically generated code. In *GPCE'08*, pp. 77–87. ACM, 2008.

[5] X. Huang. Proverb: A system explaining machine-found proofs. In A. Ram and K. Eiselt, editors, *Proc. 16th Annual Conf. Cognitive Science Society*, pp. 427–432. Lawrence Erlbaum Associates, 1994.

[6] A. Ko. Debugging by asking questions about program output. In *ICSE'06*, pp. 989–992. ACM, 2006.

[7] C. Koellmann and M. Goedicke. A specification language for static analysis of student exercises. In *ASE'08*, pp. 355–358. IEEE, 2008.

[8] S. Letovsky and E. Soloway. Delocalized plans and program comprehension. *IEEE Software*, 3(3):41–49, 1986.

[9] J. Q. Ning, A. Engberts, and W. V. Kozaczynski. Automated support for legacy code understanding. *CACM*, 37(5):50–57, 1994.

[10] RTCA Special Committee 167. Software considerations in airborne systems and equipment certification. Technical report, RTCA, 1992.

[11] Sutcliffe, G. and C. Suttner, *TPTP home page*, `www.tptp.org`.