# Generating Code Review Documentation for Auto-Generated Mission-Critical 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

*Model-based design and automated code generation are increasingly used at NASA to produce actual flight code, particularly in the Guidance, Navigation, and Control domain. However, since code generators are typically not qualified, there is no guarantee that their output is correct, and consequently auto-generated code still needs to be fully tested and certified. We have thus developed* AUTOCERT, *a generator-independent plug-in that supports the certification of auto-generated code.* AUTOCERT *takes a set of mission safety requirements, and formally verifies that the auto-generated code satisfies these requirements. It generates a natural language report that explains why and how the code complies with the specified requirements. The report is hyper-linked to both the program and the verification conditions and thus provides a high-level structured argument containing tracing information for use in code reviews.*

## 1. Introduction

Model-based development and automated code generation are increasingly used by NASA missions (e.g., Constellation uses MathWorks' Real-Time Workshop), not only for simulation and prototyping, but also for actual flight code generation, in particular in the Guidance, Navigation, and Control (GN&C) domain. However, 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. The V&V situation thus remains unsatisfactory:

- Code reviews are still necessary for mission-critical applications, but the generated code is often difficult to understand, and requires reviewers to match subtle details of textbook formulas and algorithms to model and/or code.
- Common modeling and programming languages do not allow important requirements to be represented explicitly (e.g., units, coordinate frames, quaternion handedness); consequently, such requirements are generally expressed informally and the generated code is not traced back to these requirements.
- Writing documentation is tedious and therefore often not completed or kept up to date.

In this paper, we describe a new tool that generates human-readable and traceable safety documentation from the results of an automated analysis of auto-generated code. It is based on the AUTOCERT code analysis tool [1], which takes a set of mission safety requirements, and formally verifies that the code satisfies these requirements. It can verify both simple execution-safety requirements (e.g., variable initialization before use, array out of bounds, etc.), as well as domain- and mission-specific requirements such as the consistent use of Euler angle sequences and coordinate frames. The results of the code analysis are used to generate a natural language report that explains why and how the code complies with the specified requirements. The report makes the following information explicit: assumptions on the environment (e.g., the physical units and constraints on input signals) and the intermediate variables in the computation (representing intermediate signals in the model), the algorithms, data structures, and conventions (e.g., quaternion handedness) used by the code generator to implement the model, the dependencies between variables, and the chain of reasoning which allows the requirements to be concluded from the assumptions. The analysis tool matches candidate algorithms for various mathematical operations against the code, and then uses theorem proving to check that they really are correct implementations. The report is hyper-linked to both the program and the verification conditions, and gives traceability between verification artifacts, documentation, and code. In order to construct a justification that the code meets its requirements, a diligent code reviewer must "rediscover" all the information which is automatically generated by AUTOCERT, so the high-level structured argument provided by our tool can result in substantial savings in effort.

Our approach, both to the formal verification and to the construction of the review reports, is independent of the particular generator used, and we have applied it to code generated by several different in-house and commercial code

generators, including MathWorks' Real-Time Workshop. In particular, we have applied our tool to several subsystems of the navigation software currently under development for the Constellation program, and used it to generate review reports for mission-specific requirements such as the consistent use of Euler angle sequences and coordinate frames.

## 2. Background

### 2.1. Automated Code Generation

Model-based design and automated code generation (or autocoding) promise many benefits, including higher productivity, reduced turn-around times, increased portability, and elimination of manual coding errors [2], [3]. There are now numerous 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 MathWorks' Real-Time Workshop (with the add-on product Embedded Coder), an automatic code generator that translates Simulink/Stateflow models into embeddable (and embedded) C code [4]. By some estimates, 50% of all NASA projects now use Simulink and Real-Time Workshop for at least some of their code development. Code generators have traditionally been used for rapid prototyping and design exploration, or the generation of certain kinds of code (user interfaces, stubs, header files etc.), but there is a clear trend now to move beyond simulation and prototyping to the generation of production flight code, particularly in the GN&C domain. Indeed, the prime contractor for the Orion Spacecraft (NASA's Crew Exploration Vehicle) is making extensive use of code generators for the development of the flight software.

### 2.2. Autocode Assurance

The main challenge in the adoption of code generators in safety-critical domains is the assurance of the generated code. Ideally, the code generator, itself, should be qualified or even formally verified, but this is rarely done: the direct V&V of code generators is generally too laborious and complicated due to their complex nature, while testing the generator itself can require detailed knowledge of the (often proprietary) transformations it applies [5], [6]. Moreover, the qualification is only specific to the use of the generator within a given project, and needs to be repeated for every project and for every version of the tool. Even worse, if the generator is upgraded during a project, any qualification effort which has been carried out on the previous working version is now lost, the code must be re-certified, and the entire tool-chain must now essentially be upgraded. This can offset many of the advantages of using a generator. Also, even if a code generator is generally trusted, it often requires user-specific modifications and configurations, which necessitate that V&V be carried out on the generated code [7]. In summary, the generated code still needs to be fully tested and certified.

Advocates of the model-driven development paradigm claim that by only needing to maintain models, and not code, the overall complexity of software development is reduced. While it is undoubtedly true that some of burden of verification can be shifted from code to model, there are additional concerns and, indeed, more artifacts in a model-based development process than just models. Users not only need to be sure that the code implements the model, but also 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, and so on. There can also be concerns with the understandability of the generated code. Some explanation of why and how the code satisfies the requirements, 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 documentation tool we describe here mitigates some of that risk.

### 2.3. Autocode Verification

In contrast to approaches based on directly qualifying the generator or on testing of the generated code, we have instead developed an independent autocode analysis tool which is nevertheless closely integrated with the code generator. Specifically, AUTOCERT supports certification by formally verifying that the generated code complies with a range of mathematically specified requirements and is free of certain safety violations.

However, in an *independent* V&V (IV&V) context, we must consider the larger picture of certification, of which formal verification is a part, and therefore produce assurance evidence which can be checked either by machines (during proof checking) or by humans (during code reviews). Hence, the tool constructs an independently verifiable certificate, and explains its analysis in a textual form suitable for code reviews.

If the tool does not detect any bugs, then it is guaranteed that the auto-generated source code is meets the stated requirements. Moreover, the time taken to review and certify the auto-generated code by hand, could be compared with with the time taken to do it with support from AUTOCERT.

**2.3.1. Code Analysis.** In order to certify a system, AUTOCERT is given a set of assumptions and requirements. Assumptions are typically constraints on input signals to the system, while requirements are constraints on output signals.

The tool then parses, analyzes, and verifies the generated source code with respect to the specified requirements. Note that only the code is analyzed, rather than the model or the generation process. In other words, the code generator is treated as a black box.

The key technical idea of our approach is to exploit the idiomatic nature of auto-generated code in order to automatically infer logical annotations, that is, assertions of program properties at key locations in the code. Annotations are crucial in order to allow the automatic formal verification of the requirements 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. We omit further technical details of the verification process (see [15], [8]).

During the course of verification, AUTOCERT records various facts, such as the locations of variable definitions and uses, which are later used to generate the review document (Section 4.2).

**2.3.2. Customization.** AUTOCERT is independent of the particular generator used, and need only be customized to a domain via an appropriate set of *annotation schemas*, which encapsulate certification cases for matching code fragments. We omit details of the schema language here (see [9]), but note that is is based on a generic pattern language for describing code idioms. Schemas also contain actions which construct the annotations needed to certify a code fragment, and can record other information associated with the code, such as the mathematical conventions it follows. A schema also has a number of different textual descriptions which can be parametrized by the variables in the pattern. This is used during the document generation process.

**2.3.3. Certification Browser.** The user can view the results of the verification via a *certification browser* that is integrated with Matlab. This displays the generated code along with the VCs and the review document (to be described below).

By selecting a line in the generated code, the user can see the list of VCs that are dependent on that line. The user can also select a VC and navigate to its source in the code. This action highlights the lines in the RTW-generated code which contribute to the chosen VC (that is, they had either an annotation from which the given VC was generated or contributed a safety obligation). A click on the source link associated with each VC prompts the certification browser to highlight all affected lines of code, and display the annotations for the selected VC in the RTW-generated code. Conversely, a click on the line number link at each line of code or on an annotation link will display all VCs associated with that line or annotation. A further click on the verification condition link itself displays the formula which can then be interpreted in the context of the relevant program fragments.

## 3. Mathematical Domain

We will illustrate the review document generation using excerpts that explain the verification of several requirements for an attitude module of a spacecraft GN&C system. In addition to being a necessary component of every spacecraft, the GN&C domain is challenging from a verification perspective due to its complex and mathematical nature.

We just describe the model at the top level sufficient to understand typical requirements. The attitude sub-system 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, and so on. 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, and instead are expressed informally in comments and identifier names.

At the code level, equations and transformations are expressed in terms of the usual loops, function calls, and sequences of assignments. Depending on the optimization settings of the generator, the resemblance to the model can be tenuous. Variables can be renamed and reused, and structures can be merged (e.g., via loop fusion) or split (e.g., to carry out common sub-expression elimination).

The challenge for AUTOCERT is to disentangle this complexity and provide a comprehensible explanation in terms of concepts from the model and domain (e.g., [10], [11], [12]). In effect, what the tool must do is reverse engineer the code.

In practice, this semantic abstraction can be seen as going up through several levels before reaching the high-level mathematical concepts appropriate for explanation. Fig.1 shows the relationships between these levels.

At the lowest level is the code itself along with primitive arithmetic arithmetic operators. This is, of course, the level at which V&V is actually carried out (we do not consider object code here). The purpose of comments in the code (and model) is generally to informally explain the code at a more abstract level, so AUTOCERT can be seen as formally checking these implicit conventions. At the next level are mathematical operations, such as matrix multiplication and

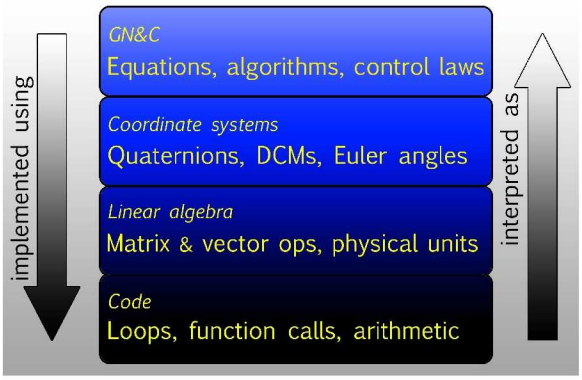


Figure 1. Levels of Abstraction

transpose, while low-level datatypes such as floats correspond, at the more abstract level, to physical values of a given unit. These, in turn, are used to represent navigational information in terms of quaternions, DCMs, Euler angles, and so in, in various coordinate systems. This is the level at which we explain the verification. There is a further level of abstraction, at which domain experts think, namely the principles of guidance, navigation, and control, itself, but explanation at this level is currently beyond our scope.

# 4. Generating Review Documents

## 4.1. Document Purpose and Assumptions

The generated safety documents serve as structured reading guides for the code and the verification artifacts, showing why and how the code complies with the specified requirements. However, the documents do not simply associate source code locations with verification conditions; in fact, we delegate this to the existing complementary code browser [1] sketched in Section 2.3.3. Instead, the documents call out the high-level operations and conventions used by the generated code (which might be different from those originally specified in the model from which the code was generated, due to optimizations) and the relevant structures in the code (in particular, the paths between the locations where the requirements manifest themselves and where they are established) and associates the verification conditions with these. This provides a "natural" high-level grouping mechanism for the verification conditions, which helps reviewers to focus their attention to the artifacts and locations that are relevant for each safety requirements, and thus conforms to the usually requirements-driven safety certification process.

The document construction is based on the assumption that all relevant information can be derived in the verification phase, in particular by the annotation inference mechanism. The document's overall structure (see Section 4.3) reflects the way the annotation inference has analyzed the program, starting with the variables occurring in the original requirements. The applied schemas implicitly also indicate which high-level conventions and operations are used by the code (see Section 4.4), and a semantic labeling of the verification conditions [13] allows us to associate only the small number of VCs with the paths that actually contribute to demonstrating how a given requirement holds along a path, as opposed to those that are just coincidentally related to it (see Section 4.5).

## 4.2. Technical Approach

The generated documents are heavily cross-referenced and hyper-linked, both internally and externally, so that HTML/JavaScript is a suitable technical platform. Cross-linking follows not only from the hierarchical document structure (e.g., the links from the requirements summary to the individual requirements sections, see Fig. 2), but also from the traceability links recovered by the analysis phase, primarily the chains of implications from the properties of one variable to the properties of one or more "dependent" variables. Hyper-links are mostly traceability links to other artifacts such as external documents, models, code, or verification conditions that were constructed by by the analysis and verification phases. Further hyper-links can be introduced by the concept lexicalization; these usually refer to to external documents such as RTW documentation or Wikipedia pages.

The actual document generation process is relatively lightweight and does not require the application of deep natural language generation (NLG) technology [14]. Currently, the document's overall structure is fixed, so that content determination and discourse planning are not necessary. Concept lexicalization, however, relies on text fragments provided by the annotation schemas (for the mathematical and data structures and the operations) or stored in a fact base (for the mathematical operations used in assumptions and other formulas). This step can thus be customized easily.

The document generator contains canned text for the remaining fixed parts of the document, and constructs some additional "glue text", to improve legibility. The combined text is post-processed to ensure that the document is syntactically correct. The generator currently directly produces HTML, but changing the final output to, e.g., XML to simplify layout and rendering changes is relatively straightforward.

## 4.3. Document Structure

The document consists of a general introduction and a section for each certified requirement. The introduction contains a natural language representation of the formalized requirements and certification assumptions; see Fig. 2 for an

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:

- `rty_7` is a value representing Mach at MSL altitude
- `rty_2` is a value representing position in the ECI frame
- `rty_1` is a value representing velocity in the ECI frame
- `VelocityCompNed` is a value representing velocity in the NED frame

The assumptions for the certification are that

- `BitwiseOperator_c` is positive
- `VelocityNED_e` is a value representing velocity in the NED frame
- `DCMtoQuat_1` is a quaternion representing a transformation from the NED frame to the body fixed frame (Body)
- `AtmScaleHt_MslAlt` represents the altitude entries in a lookup table
- `SpeedOfSound_Lookup` represents the speed of sound entries in a lookup table
- `GeodeticHeight_g` is a value representing geodetic height
- `Latitude_g` is a value representing geodetic latitude
- `Longitude_a` is a value representing longitude
- `rty_11` is a value representing altitude
- `rty_12` is a value representing angular velocity

Figure 2. Requirements and Assumptions

example.[1] This allows the reviewers to check that the formalization has not (inadvertently) introduced any conceptual mismatches. The verbalization is based on an analysis of the formula structure, and uses text templates to verbalize the relevant predicates. This allows us to customize the document's appearance.

The requirements sections are automatically grouped into categories which correspond to the applied logic (i.e., the safety policy [15]); this information can be derived from the structure of the given formalization of the respective requirements. Each requirement section in turn starts with a summary of the pertinent information, i.e., the relevant variables and the high-level conventions and operations used by the code (see Section 4.4). The system extracts from the given formalization the program variables that correspond to the signals for which the requirement has to hold, and then identifies the intermediate variables (mostly corresponding to intermediate signals in the model) that form the chain between the program locations where the requirement holds and where it is established. The document separately lists both the initial and the intermediate variables. However, the system discards variables for which the formal proof is below a certain threshold of complexity. This reduces the lists to those variables to which reviewers need to direct their attention.

1. For presentation purposes, we converted the excerpted HTML document fragments into LaTeX, but kept their structure and text; to improve legibility, we also removed most HTML links, in particular those associated with source code references and those introduced by the concept lexicalization.

Each requirements section then concludes with a series of subsections that explain why and how each of the relevant variables meets the requirement (see Section 4.5). The subsections can contain explanations of fragments of code, and can refer to the explanations for other variables, which are cross-linked. Whenever the underlying certification tool has carried out some analysis using the prover (e.g., that a code fragment establishes some property), the document provides links to the corresponding verification conditions (see Section 4.6).

## 4.4. Inferred Operations and Conventions

As part of its analysis, AUTOCERT effectively "reverse engineers" the code, and identifies the potentially overlapping fragments that correspond to high-level operations specified in the model. As a side effect of this analysis, AUTOCERT also 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 four scalars representing a quaternion.

The report contains a concise summary of this information, going from the abstract mathematical structures to the the concrete operations; see Fig. 3 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 next to each other. This 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.[2] Note that here we choose to list the case where a high-level mathematical structure's representation is distributed over several variables (i.e., `eml_fv5`, `eml_fv6`, and `eml_fv7`), but not to list all the program variables and what they represent, since the reuse of variables by optimizing generators makes this aspect less useful. However, both decisions could easily be changed by simply changing the schemas.

## 4.5. Explaining Inferred Program Structure

The backbone of the document is a chain of implications from the properties of one variable to the properties of one

2. Note that different representations are not necessarily unsafe or unwanted (in fact, DCMs and quaternions can represent the same information), but might nevertheless indicate deeper design problems.

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.
- DCMs are represented as three 3-vectors.
- The vectors `eml_fv5`, `eml_fv6`, and `eml_fv7` together represent a DCM.
- Quaternions are right-handed.

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

- a coordinate transformation using a DCM from ECI to ECEF
- a coordinate transformation using a DCM from NED to ECEF
- a coordinate transformation using a DCM from NED to Nav
- conversion of a DCM to a quaternion
- conversion of a quaternion to a DCM
- matrix multiplication
- matrix transpose

Figure 3. High-level Conventions

The variable `T_NED_to_body1` has a single relevant occurrence at line 235 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 three verification conditions:

- Attitude_frame_016_0025 (i.e., establish the postcondition at line 235 (#1))
- Attitude_frame_016_0026 (i.e., establish the postcondition at line 235 (#2))
- Attitude_frame_016_0027 (i.e., establish the postcondition at line 235 (#3))

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 `Util_Matrix_Multiply`, as above. 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 235 in file `Attitude.cpp` is connected to the establishing location at lines 200 to 203 in file `Attitude.cpp` by a single path, which, beginning at this location, runs through the next six statements, starting with the procedure `Util_DCM_to_Quat` at line 205 in file `Attitude.cpp`, before it calls the procedure `Util_Matrix_Multiply` at line 230 in file `Attitude.cpp`. This path gives rise to two verification conditions:

- Attitude_frame_018_0031 (i.e., establish the postcondition at line 226 (#1))
- Attitude_frame_018_0032 (i.e., establish the postcondition at line 226 (#2))

Figure 4. Uses and Paths: A Step in the Argument

or more "dependent" variables. The chain starts at those key variables which appear in the requirement, and continues to

The frame safety is established at a single location, lines 177 to 189 in file `Attitude.cpp` by definition as a DCM matrix from NED to NAV. The correctness of the definition gives rise to two verification conditions:

- Attitude_frame_006_0009 (i.e., establish the postcondition at line 189 (#1))
- Attitude_frame_007_0010 (i.e., establish the precondition at line 177 (#1))

Figure 5. Definitions

variables in the assumptions or input signals. Fig. 4 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 requirement properties. Three 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. In this case, it turns out that that particular definition has been explained earlier in the document, so a link is given to the relevant section (cf. "as above"). We give an example of a definition below. 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.

**Explaining the definitions.** Fig. 5 gives an example where a DCM has been identified and verified. It gives links to the appropriate lines in the code and links to the VCs that demonstrate the correctness of the definition. In this case there are two VCs: a pre-condition (omitted here), which states that there exist heading and azimuth variables, and a post-condition, which states that the constructed matrix does indeed satisfies the textbook definition of a DCM from Ned to NAV, with entries equivalent to the appropriate trigonometric expressions. Structures that involve loops generally have considerably more correctness conditions, with VCs for inner and outer invariants, as well as pre- and post-conditions.

## 4.6. Tracing

The provision of traceability links between artifacts is crucial to providing certification support since things cannot be understood in isolation. Indeed, the code review document generated by AUTOCERT can be seen as a structured high-level overview of the traceability links inferred during verification. There are both *internal* links, where items within

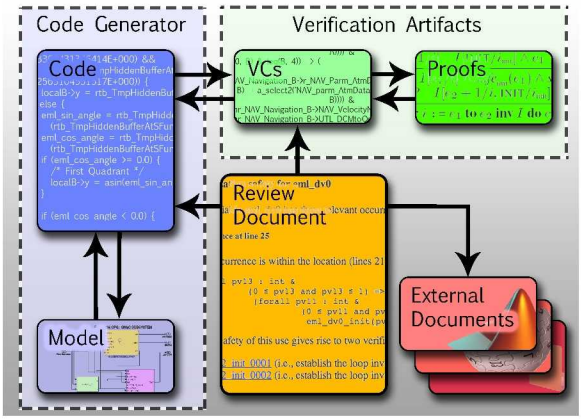


Figure 6. Tracing Between Artifacts

the document are linked to each other, and *external* links to other artifacts.

The internal links have been described above, and include links from requirements to safety policies, variables, and concepts. Fig. 6 illustrates the different kinds of external tracing provided by AUTOCERT within the larger Matlab environment. Matlab/RTW already provides bidirectional linking between models and code. To this, the AUTOCERT certification browser adds bidirectional linking between code and VCs. The review documents provide a further layer of tracing, linking code, VCs, and external documents such as Matlab block documentation and Wikipedia articles on domain concepts.

## 5. Conclusion

We have described the review documentation feature of AUTOCERT, an autocode certification tool which has been customized (but is not limited) to the GN&C domain, and have illustrated its use on code generated by Real-Time Workshop from a Matlab model of an attitude sub-system.

AUTOCERT automatically generates a high-level narrative explanation for why the specified requirements follow from the assumptions and a background domain theory, and provides hyperlinks between steps of the explanation and the relevant lines of code, as well as the generated verification conditions.

The tool is aimed at facilitating code reviews, thus increasing trust in otherwise opaque code generator 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. 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.

Much more can be done to improve the review documents themselves, such as adding more hierarchy and top-level summaries, and listing formulas and equations that are used in the code. In particular, more information could be gleaned from the proofs, such as the use of constants, lookup tables, as well as the specific assumptions and axioms used by individual requirements, and whether there are any unused assumptions. However, we are already working on such a proof analysis, and foresee no particular problems in extending the document generator accordingly. We also continue to extend the underlying domain theory that is used to verify the code.

**Acknowledgments.** Thanks to Allen Dutra for help with the graphics.

## References

[1] E. Denney and S. Trac, "A software safety certification tool for automatically generated guidance, navigation and control code," in *IEEE Aerospace Conference Electronic Proceedings*. Big Sky, Montana: IEEE, 2008.

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

[3] A. Kleppe, J. Warmer, and W. Bast, *MDA Explained. The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

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

[5] I. Stürmer and M. Conrad, "Test suite design for code generation tools," in *Proceedings of 18th IEEE International Conference on Automated Software Engineering*. IEEE, Oct. 2003, pp. 286–290.

[6] I. Stürmer, D. Weinberg, and M. Conrad, "Overview of existing safeguarding techniques for automatically generated code," *SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–6, Jul. 2005.

[7] T. Erkkinen, "Production code generation for safety-critical systems," MathWorks, Tech. Rep., 2004.

[8] E. Denney and B. Fischer, "A generic annotation inference algorithm for the safety certification of automatically generated code," in *Proceedings of the Conference on Generative Programming and Component Engineering (GPCE '06)*. Portland, Oregon: ACM Press, October 2006, pp. 121–130.

[9] ——, "Generating customized verifiers for automatically generated code," in *Proceedings of the Conference on Generative Programming and Component Engineering (GPCE '08)*. Nashville, TN: ACM Press, October 2008, pp. 77–87.

[10] D. A. Vallado, *Fundamentals of Astrodynamics and Applications*, 2nd ed., Space Technology Library. Microcosm Press and Kluwer Academic Publishers, 2001.

[11] J. Diebel, "Representing attitude: Euler angles, unit quaternions, and rotation vectors," Stanford University, Tech. Rep., Oct. 2006.

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

[13] E. Denney and B. Fischer, "Explaining verification conditions," in *12th International Conference on Algebraic Methodology and Software Technology (AMAST 2008)*, Urbana, Illinois, July 2008.

[14] E. Reiter and R. Dale, *Building Natural Language Generation Systems*. Cambridge University Press, 2000.

[15] E. Denney and B. Fischer, "Correctness of source-level safety policies," in *Proceedings of FM 2003: Formal Methods*, Lecture Notes in Computer Science, vol. 2805. Pisa, Italy: Springer, Sep. 2003, pp. 894–913.