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

Model-based development and automated code generation are increasingly used for production code in safety-critical applications, but since code generators are typically not qualified, the generated code must still be fully tested, reviewed, and certified. This is particularly arduous for mathematical and control engineering software which 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. Both tasks are complicated by the often opaque nature of auto-generated code. We address these problems by developing a verification-driven approach to traceability and documentation. We apply the AUTOCERT *verification system to identify and then verify mathematical concepts in the code, based on a mathematical domain theory, and then use these verified traceability links between concepts, code, and verification conditions to construct a natural language report that provides a high-level structured argument explaining why and how the code uses the assumptions and complies with the requirements. We have applied our approach to generate review documents for several sub-systems of NASA's Project Constellation.*

1. Introduction

Model-based development and automated code generation have moved beyond simulation and prototyping and are increasingly used for actual production code generation, in particular in mathematical and engineering domains. For example, NASA's Project Constellation uses MathWorks' Real-Time Workshop for its Guidance, Navigation, and Control (GN&C) systems. However, code generators are typically not qualified [24] and there is no guarantee that their output is correct, so that the generated code must still be fully tested and, for safety-critical applications, reviewed and certified. This is difficult for two reasons in particular:

- 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 modelling and programming languages do not allow important domain requirements to be represented explicitly (e.g., units, coordinate frames, quaternion handedness [16], [28]); consequently, such requirements are generally expressed informally and the generated code is not traced back to them.

The central problem for reviewing and certification is to disentangle the complexity of the generated code, in order to provide a comprehensible explanation in terms of high-level domain concepts. This in turn requires comprehensive traceability links that connect the code not only to the model or to verification artifacts, but also to abstract concepts and requirements such as quaternion handedness. The central challenge is to recover these traceability links: we cannot rely on the code generator to provide them, and, furthermore, we cannot even rely on the correctness of any links that are provided. In fact, we need explicit assurance that the traceability links–whether provided or recovered—are correct, because any documentation derived from them would be misleading otherwise, and nothing would be gained.

In this paper, we thus describe a new verification-driven approach to traceability and documentation. Our goal is to to construct verified, natural language safety documentation that explains why and how automatically generated code complies with specified requirements. The central insight of our approach is that verification and documentation need the same links, and that we can combine methods from program understanding and program verification to recover *verified traceability links*. We use the recovered links to construct documentation that lists and explains 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.

The documentation generation tool described here is based on the AUTOCERT code analysis tool [5], [6], which takes a set of requirements, and uses a Hoare logic approach to formally verify that the code satisfies them. AUTOCERT can verify both execution-safety requirements (e.g., variable initialization before use, array bounds, etc.), as well as domain- and mission-specific requirements such as the consistent use of Euler angle sequences and coordinate frames. Here we reuse two of AUTOCERT's basic components, the annotation inference algorithm [5], [6] and the schema compiler [8]. However, we significantly improve over this infrastructural basis. In particular, the very ideas of verified traceability link recovery and generating safety documentation are new. In addition, this work also required substantial changes to the schema compiler and the development of the proof analysis and the document generation.

Tools such as the MathWorks' Model Advisor do provide support for inspections at the model level, but formal approaches to certification and inspection of source code are much less common. There has been much work on recovering high-level traceability information from both formal artifacts, such as code [1], and informal artifacts using probabilistic methods [20]. Much of this work, however, is aimed at program comprehension rather than certification and we are not aware of any verifiable approach.

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. AUTOCERT

Generator Assurance. The many benefits promised by model-based design and automated code generation, such as higher productivity and elimination of coding errors [3], [13] can only be realized if the generated code can be assured to be correct. Ideally, code generators should be formally verified, but due to their complex nature this is generally too laborious and complicated. Testing and qualifying the generator can require detailed knowledge of the (often proprietary) transformations it applies [26]. Moreover, qualification is limited 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. Also, even if a code generator is generally trusted, it often requires user-specific modifications and configurations, which still require that the generated code is fully tested and certified [10].

Product-Oriented Certification. In contrast to approaches based on directly qualifying the generator or on testing of the generated code, we follow a product-oriented approach, in which every generated program is certified individually. In order to certify a program, AUTOCERT is thus given a set of formal assumptions and requirements. Assumptions are typically constraints on input signals to the system, while requirements are constraints on output signals. AUTOCERT then formally verifies that the generated code complies with the specified requirements. It is implemented as a generator plug-in, but since it only analyzes the code and not the model or the generation process, the generator remains a black box.

However, certification requires more than black box verification of selected properties, such as an explanation of why and how the code satisfies the requirements, otherwise trust in one tool (the generator) is simply replaced with trust in another (the verifier). The work presented here addresses this problem.

Code Analysis. We use automated theorem provers (ATPs) to verify the requirements; however, to achieve full automation without access to the code generator internals, we need *annotations*, i.e., logical assertions of program properties, at key locations in the code. Hence, we split certification into an untrusted annotation construction phase that uses a post-generation inference technique, and a simpler but trusted verification phase, where the standard machinery of a verification condition generator (VCG) and ATP is used to prove that the code satisfies the required properties.

The annotation inference exploits the idiomatic nature of auto-generated code and is driven by a generator- and property-specific set of idioms. We distinguish several classes of idioms, in particular *uses*, which refer to locations where the property is required (i.e., a requirement materializes), and *definitions*, where the relevant properties are ultimately established. The inference algorithm builds an abstracted control-flow graph (CFG), using the patterns to collapse the code idioms into single nodes. It then traverses the CFG from variable use nodes backwards to all corresponding definitions and annotates the statements along the paths as required. For further technical details of the verification process omitted here see [4], [6].

Browsing. The user can view the verification results via a *certification browser* [9] integrated into Matlab. This displays the generated code along with the verification conditions (VCs) and the review document, and allows users to trace between lines of code and associated VCs, as well as code and concepts.

Customization. AUTOCERT is independent of the particular generator used, and need only be customized to a domain via

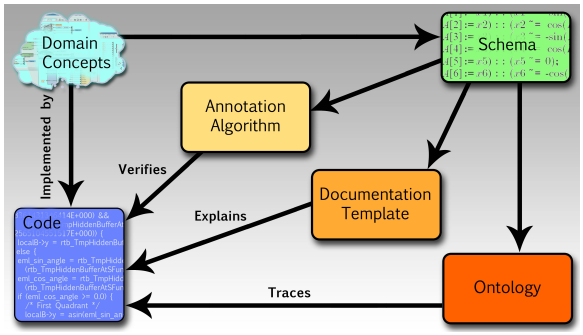

Figure 1. From concepts to documentation

an appropriate set of *annotation schemas*, which encapsulate certification cases for matching code fragments; we will give examples of schemas in the next section. The schemas use a generic pattern language to describe code idioms. They 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 textual description which can be parametrized by the variables in the pattern, and slots for recording concepts. These are used during the document generation process. Hence, schemas are central to achieving our goal of a unified approach to verification, documentation, and tracing.

An annotation schema compiler (see [8]) takes a collection of annotation schemas tailored towards a specific code generator and domain, and compiles it down into customized annotation templates, documentation templates, and concept relations drawn from a domain ontology. The annotation templates are then applied using a combination of planning and aspect-oriented techniques to produce an annotated program. Figure 1 illustrates this use of schemas for multiple purposes in the code assurance process.

3. Encoding the Domain Knowledge

3.1. Guidance, Navigation, & Control Domain

We will illustrate our approach over the GN&C domain, which is challenging from a verification perspective due to its complex and mathematical nature; more precisely, we use the verification of several requirements for an attitude subsystem of a spacecraft GN&C system, which is a necessary (and safety-critical) component of every spacecraft.

The attitude sub-system takes several input signals that represent different physical quantities, and computes output signals representing other quantities, such as Mach number or angular velocity and position. Signals are generally represented as floating point numbers or as quaternions and have an associated physical unit and/or frame of reference. Here we consider the vehicle-centered systems North-East-Down (NED) and wander azimuth (Nav), and the earth-centered systems Earth-Centered Inertial (ECI) and Earth-Centered Earth Fixed (ECEF).

$$\begin{pmatrix} \text{-}\cos\lambda\,\sin\phi & \text{-}\sin\lambda & \text{-}\cos\lambda\,\cos\phi \\ \text{-}\sin\lambda\,\sin\phi & \cos\lambda & \text{-}\sin\lambda\,\cos\phi \\ \cos\phi & 0 & \text{-}\sin\phi \end{pmatrix}$$

$$\begin{pmatrix} \cos(H{-}A) & \sin(H{-}A) & 0 \\ \text{-}\sin(H{-}A) & \cos(H{-}A) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Figure 2. DCM matrices: (top) *NED-to-ECEF* (bottom) *NED-to-Nav*

At the conceptual level, a transformation between two different frames can be represented by a direction cosine matrix (DCM) [28]; Figure 2 shows the different structure of two example DCMs transforming from the NED frame into two different target frames. However, GN&C systems usually use quaternions to represent frames, so that, even at the model level, these frame transformations involve converting the quaternions to DCMs, applying some matrix algebra, and then converting them 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).

For the certification of *frame* safety (i.e., all measurements are transformed into the right frames before they are processed) we need to be able to distinguish between different DCMs, but the code generated by Real-Time Workshop uses temporary variables to store the elements, and the matrix (represented as a vector) is updated using these (see Figure 3). Note that additional temporaries are used to factor out common subexpressions. In order to identify the sequence of array updates as the *DCM-NED-to-Nav* idiom, and to distinguish it from the structurally equivalent *DCM-NED-to-ECEF* idiom, we thus need to analyze the *content* of the variables on the right-hand sides.

Another example is the equation for calculating the Mach number in terms of velocity and speed of sound at a given altitude, which is given by $M_a = V/S_a$, where V is velocity and S_a is the speed of sound at altitude a. Although this is a simple equation, the connection to the code is far from obvious. Figure 4 shows the relevant fragments. In particular, because of various optimizations, the code is distributed

```
c0:=0
c1:=1
...
w0:=in6-in7;
w1:=sin(w0);
w2:=cos(w0);
...
a[0]:=w2;
a[1]:=v1;
a[2]:=c1;
...
a[8]:=c1;
```

Figure 3. *DCM-NED-to-Nav* code fragment

```
247 LookUp_real32_T_real_Treal32_T(
248   &(rtb_LookupSoS),
249   &AtmDataSoS_l[0],
250   NAV_MslAltitude,
251   &AtmDataSoS_u[0], 4U);
...
438   for(k=0; k<3; k++) {
...
452     V_body[k]=NAV_VelocityNED[k];
453   }
...
455   sin_theta=norm((real32_T *)V_body);
...
563   rtb_NAV_MachNum=sin_theta/rtb_LookupSoS;
...
624 NAV_MachNum=rtb_NAV_MachNum;
```

Figure 4. Code fragment matching `mach_ned`

widely (or *delocalized* in the terminology of [17]). In the code, the velocity is, of course, a vector, and must be given in a specific frame of reference, in this case NED. The code must also take the vector norm when computing the Mach number. The lookup function is used to determine the speed of sound at a given altitude above mean sea level (MSL). It also takes pointers to tables of data that give the function between altitude and speed of sound at various altitudes (the actual value is calculated by interpolation). Note also the misleading identifier `sin_theta` which is simply reused from elsewhere in the code. Misleading names such as this would prove a problem for IR-based traceability techniques.

These examples already show that the main verification challenge is to disentangle code-level complexity and to provide a comprehensible explanation in terms of higher-level domain concepts. In practice, this semantic abstraction can be seen as going up through several levels before reaching the high-level mathematical concepts (e.g., in the language of [16], [28]) appropriate for explanation. Figure 5 shows the relationships between these levels.

At the lowest level is the code itself along with primitive arithmetic operators. This is, of course, the level at which a code review is actually carried out. The purpose of

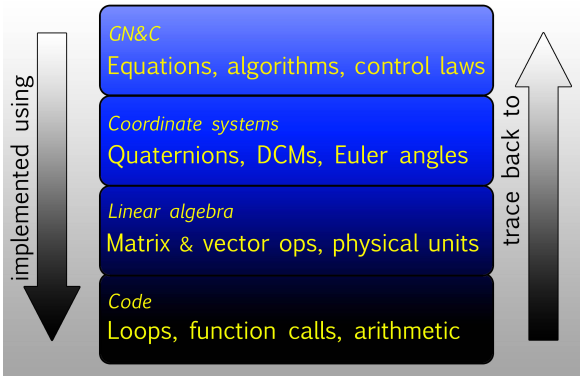

Figure 5. Levels of domain abstraction

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 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 etc., 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.

3.2. Schemas

Annotation schemas characterize a domain at the implementation level. They have the following elements: name and description (parametrized by pattern variables) used in the safety document, policy, pattern category, annotation pattern, matching condition (whether the pattern must be matched exactly or whether intervening "junk code" can appear), dependent variables (i.e., variables used by the pattern which require further semantic information, hence triggering further inference), pre- and post-actions (executed before and after annotation), and concept list. Some of these elements are optional and not shown in the examples. Actions can modify the inference state and include looking up and asserting extra-logical information used during inference, as well as computation of annotations. The concept list gives a list of concept tokens (which can be parametrized by pattern variables) associated with the schema, and is used to generate facts later used in the tracing and documentation.

Figure 6 shows the schema used to capture the *DCM-NED-to-Nav* idiom. It uses a specific constraint operator ~= that triggers a simple, approximate data-flow analysis to infer possible symbolic values of program variables that are then checked against the constraint pattern. During verification, the precondition induces VCs that verify these

```
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 has_frame(V, dcm(ned, nav)),
        pre ∃ ψ, φ · has_unit(ψ, heading)
            ∧ has_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 6. Annotation schema `dcm_ned_nav`

```
schema(mach_ned=['Mach computation in', M]
, frame
, def(M)
, (M:=y :: (y ~= norm(V)/S, V :: vel(ned), S :: sos)
  ) ← &(post has_unit(M, mach(alt)))
, [V, S]
, [eqn=mach(M,V,S), unit=vel(V), frame=ned(V)]
).
```

Figure 7. Annotation schema `mach_ned`

values and thus the results of the approximate analysis. Hence, we get a proven match. The concept list states that the matched code is associated with a DCM representation as a 9-vector.

Figure 7 shows the schema for the Mach computation. It is similar to the original equation, but explicit the constraints on the variables explicit, and declares the associated concepts, recording that the Mach-equation was instantiated for the variables M, V, and S, and that V has unit velocity, and frame NED. AUTOCERT will match this against the delocalized fragments given in Figure 4, inserting annotations after the assignments to M, V, and S. After matching M, the annotation algorithm continues on recursively to generate annotations for the constraints on the dependent variables V and S, as well as on any intervening code that requires annotation. For example, the loop where `V_body` is assigned `NAV_VelocityNED` will be given an invariant and post-condition. This is all handled automatically, and need not be stated in the schema.

The DCM and Mach examples illustrate one particular kind of complexity. Another kind, not shown here, comes from loops. Our schema language also supports the analysis of (nested) loops, and generates the necessary loop

```
fof(transformation_composition, axiom,
    ! [F1, F2, F3, M1, M2] :
    ((has_frame(M1, dcm(F1,F2)) &
      has_frame(M2, dcm(F2,F3)))
     => (has_frame(mmul(M2,M1), dcm(F1,F3))))
).

fof(transformation_application, axiom,
    ! [F1, F2, M1, M2] :
    ((has_frame(M1, F1) &
      has_frame(M2, dcm(F1,F2)))
     => has_frame(mmul(M2,M1), F2))
).

fof(transformation_dcm_quat_equiv, axiom,
    ! [F1, F2, D, Q] :
    ((has_frame(D, dcm(F1,F2)) &
      dcm_equiv_quat(D, Q))
     => has_frame(Q, quat(F1,F2)))
).
```

Figure 8. Typical Axioms of the Domain Theory

invariants, and we refer to [8] for examples. Note that the schema complexity does not "bleed through" to the generated documents, because these rely on the concepts.

3.3. Axioms

Axioms provide the logical formalization of a domain, and are the basis for formal proofs. There are about 30 axioms in the theory of coordinate systems, consisting of definitions for the various transformation matrices and quaternions, and their interaction with the operators of linear algebra, and roughly the same number again for each of linear algebra and elementary arithmetic. Some examples are given in Figure 8, which uses the standard TPTP notation [27]. There are more elaborate axioms, however, with nested quantifiers, so we do need the full power of automated theorem provers. There is also a smaller theory giving the relevant laws of physics (such as the Mach equation) in logical form. The programming language is not formalized axiomatically, but rather via the VCG.

4. Verified Traceability Link Recovery

4.1. Link Categories

In model-based development, the concept of traceability is generally equated with maintaining or recovering links between model and code, or more precisely, from the individual elements of the model (e.g., Simulink boxes) to their representation in the generated code. Most commercial code generators add origin information directly to the generated code (usually as comments or embedded hyperlinks), and academic research has worked on recovering these links after model or code changes (see, e.g., [1], [23]).

However, this is not sufficient for our purposes, for two different reasons. 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 also need to explain the *internal* conceptual structuring 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. Hence, *internal traceability links* are required to relate different code locations to each other.

We can distinguish several link categories, depending on the entities related to each other. However, since certification is driven by the requirements, the links to and from these are more important.

Requirement-to-concept links relate the individual requirements to the concepts in the three upper tiers of the domain abstractions (see Figure 5). 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 are commonly used to certify 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.

Requirement-to-axiom links are similar but relate the requirements to the specific domain theory axioms that are used by the ATP to prove the VCs associated with the requirement. Note that most ATPs treat assumptions and axioms interchangeably, but in the certification they play different roles. The assumptions are specific to the code, and need to be established by other system components, while the axioms are "hardened" over time and 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 previous categories, since the VCs give access to the proofs.

A number of complementary code-based links are used to compute the requirements-based traceability links. Other links, for example code-to-concept, can be derived from those defined here, as can be the reverse links. However, not all of those are currently supported by our implementation.

Code-to-code links reflect the internal *conceptual* structure of the code, not its syntax: two code locations are linked if they are connected in the abstracted CFG built by the annotation inference.

Simple **code-to-VC** links are provided by most formal verification tools but more refined 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 VCs fails.

4.2. Link Recovery via Annotation Inference

The core traceability links required to generate meaningful documentation are requirement-to-concept and code-to-code links. Both are recovered by AUTOCERT's annotation inference. The code-to-code links are recovered as side effect of the CFG traversal, but the crucial fact that allows the recovery of requirement-to-concept links here is that, in effect, the already schema tells us everything we need to know about the code. This is based on the insight that matching a schema against the code is actually performing program understanding, and that the schema application itself thus already reflects all domain concepts that can be extracted from the matched code fragment. 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* idiom again. If the schema shown in Figure 6 is matched against the code in Figure 3, and all VCs are proven, then we know for certain that the code is related to the *DCM* concept and, in particular, represents a DCM as a 9-vector. 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.

4.3. Link Recovery via Verification

The verification machinery also contributes to link recovery. Since the VCG processes the code, it is primarily responsible for the code-to-VC links. It can add 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 labelling [7] 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 are currently only analyzing proofs in the standard TPTP proof notation [27], which restricts us to using the ATPs E [25] and SPASS [30]. Since AUTOCERT generates the complete proof tasks, we can rely on naming conventions under its control to distinguish between assumptions and axioms.

4.4. Tracing

The recovered traceability links between artifacts allow a higher level of certification support. Rather than just

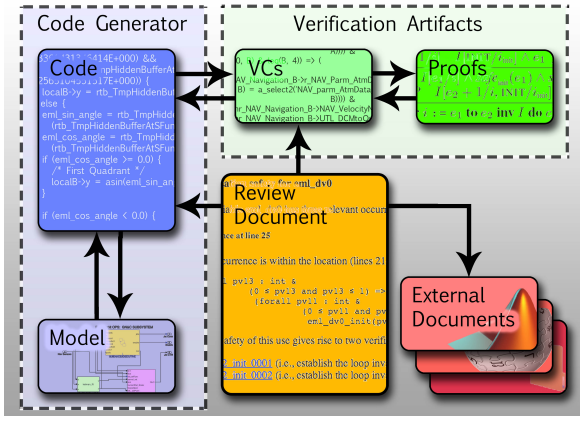

Figure 9. Tracing Between Artifacts

verifying the requirements we can trace the requirements to the code, as mandated by DO-178B [24] and NASA-internal standards (NPR 7150.2), but also to assumptions and VCs.

Indeed, the code review document generated by AUTOCERT can be seen as a structured high-level overview of the traceability links recovered during verification. Figure 9 illustrates the different kinds of 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. Generating Review Documents

5.1. Documentation Purpose and Principles

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. However, the documents do not simply associate source code locations with VCs; in fact, we delegate this to the existing complementary code browser [9] sketched above. 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 VCs with these. 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, and thus conforms to the usual requirements-driven certification process.

The document construction is based on the principle that, once the assumptions and requirements have been specified, all relevant explanatory information is reflected by the traceability links recovered in the verification phase, in particular by the annotation inference mechanism. The document's overall structure (see Section 5.2) 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 5.3), and the semantic labelling of the VCs [7] 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 5.4).

5.2. 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 Figure 10 for an example.[1] This allows the reviewers to check that their 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.

The introduction also represents the requirement-to-assumption links recovered by the proof analysis, and calls out assumptions that are not necessary (see Figure 10 again).

The requirements sections are automatically grouped into categories which correspond to the particular logic which is applied (i.e., the safety policy [4]); 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 5.3). 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.

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 to the body frame
...
5) rty_3 is a quaternion representing a transformation from the ECI to the NED frame
6) rty_7 represents Mach at MSL altitude

The assumptions for the certification are that

1) 7.29212e-05 represents angular velocity, which is used in requirements 3 and 4.
2) PadLongAtLaunch represents a longitude.
3) PadLatAtLaunch represents a geodetic latitude.
4) BitwiseOperator_c is positive, which is used in requirements 1, 2, 3, 4, and 5.
5) AtmScaleHt_MslAlt is a table of entries for altitude, which is used in requirement 6.
6) TrueHeading_h represents a true heading, which is used in requirements 1, 2, 3, and 4.
7) PlatformAzimuth_j represents a platform azimuth, which is used in requirements 1, 2, 3, and 4.
8) AttitudeBodyToNav_o is a quaternion representing a transformation from the body to the Nav frame, which is used in requirements 1 and 3.

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

2) PadLongAtLaunch represents a longitude.
3) PadLatAtLaunch represents a geodetic latitude.

Figure 10. Requirements and Assumptions

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 5.4). 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 VCs (see Section 4.4).

5.3. Inferred Operations and Conventions

As part of its analysis, AUTOCERT effectively "reverse engineers" the code, and 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 vectors representing a DCM.

The report contains a concise summary of this information as represented by the recovered traceability links, going from the abstract mathematical structures to the the concrete operations; see Figure 11 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 sub-categorization is derived from the 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.[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 description lists in the schemas.

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 11. High-level Conventions

5.4. 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 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 12 shows one step in this chain.

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 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 12. Uses and Paths: A Step in the Argument

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. 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. Figure 13 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

The frame safety of `Reshape9to3x3columnmajor_o` 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 13. Explaining Definitions

The proofs of requirement 1 use the following assumptions

4) `BitwiseOperator_c` is positive
6) `TrueHeading_h` represents a true heading
7) `PlatformAzimuth_j` represents a platform azimuth
8) `AttitudeBodyToNav_o` is a quaternion representing a transformation from the body to the Nav frame

and the following elements of the domain theory:

- definition of a DCM from NED to Nav
- arithmetic reasoning
- composition of transformation of frames
- preservation of frames under conversion of a DCM to a quaternion
- preservation of frames under conversion of a quaternion to a DCM
- transposition of frames under matrix transpose

Figure 14. Proof Analysis Results

demonstrate the correctness of the definition. In this case there are two VCs: a pre-condition, 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 (cf. Figure 6). Structures that involve loops generally have considerably more correctness conditions, with VCs for inner and outer invariants, as well as pre- and post-conditions.

5.5. 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 [12]. We thus only present information from the corresponding traceability links, but compress this even further. First, we tag each axiom with a category (e.g., representing arithmetic reasoning) and only list the categories. Second, we combine the output of entire VC sets, again using recovered traceability links to identify conceptually related VCs.

Figure 14 shows how we summarize the proofs for all VCs corresponding to a requirement. Here, all arithmetic reasoning is hidden under a single entry, but the more relevant axioms representing frame reasoning are individually tagged

as categories and thus listed individually. Since the axiom names are internal and convey no meaning to a reviewer, we associate explanatory texts with the categories.

5.6. 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 Figure 10), 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 VCs that were constructed 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 [22]. 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 produces directly HTML, but changing the final output to XML to simplify layout and rendering changes is straightforward.

6. Preliminary Evaluation

This work can be evaluated in two different ways. First, we can ascertain the degree to which the generated documents and traceability links help reviewers understand the code during code reviews; however, qualified domain experts are hard to come by, so we have primarily anecdotal feedback here. Second, we can measure the accuracy of the technique itself in terms of the recovered links.

In an initial experiment, we showed the generated code to a reviewer, along with the original models and the requirements. The subject is familiar with code reviews and the general GN&C domain, but not a domain expert. We first asked the subject to manually trace a subset of the code to the model, and to justify why some of the requirements hold. Both tasks turned out to be very hard, due to optimizations, in particular the reuse of identifiers for different purposes, and the subject gave up after four hours without finishing the task. We then asked the subject to repeat the task for a different subset of code and requirements, but this time with the help of the generated documentation. The subject found that the generated documents simplified the overall tracing task but that tracing from code to concepts was not supported well enough by our prototype. The subject also suggested to add more hierarchical structure to the documents, and a complementary, forwards-oriented documentation style (i.e., from assumptions to requirements). Overall, however, the subject confirmed the general approach.

In contrast to IR-based approaches to traceability, our verification-based approach is inherently exact, and so traditional notions of precision and recall are not appropriate means of evaluation. Instead, we evaluate the *coverage* of explanations and VCs, analogously to test case coverage.

First, we analyzed unoptimized code (i.e., generated with the optimization settings off), which comprised 201 non-blank, non-comment lines. We certified five requirements contributing to frame safety, which produced 80 VCs and a document of 38 pages (measured as the length of the HTML default printout). All 80 VCs were proven by a combination of the ATPs E (V0.999) [25] and SPASS (V3.0c) [30]. Of course, in general we will not achieve perfection, because the domain theories can be incomplete, the annotation schemas will contain bugs, and provers must work with finite resources.

We analyzed two of the five requirements. In both cases, the relevant part of the document was 7 pages. For each requirement, we manually determined the relevant fragments of code, which was 50 and 100 LOC, respectively. We then computed the VC and document coverage by counting up which code lines were traced to. For the first requirement, the 11 VCs gave 100% coverage. In other words, each line had a corresponding logical definition. The document explained all but 14 lines, which gives a coverage of 72%. The omission is because indirections in assignments do not get explained (but they do get verified; in fact, the explanation of the VCs—see [7]—traces to these missing lines as well). For the second requirement, the VC coverage was again 100%, but the document coverage was lower, 37%, for the same reason. In general, we do not expect full VC coverage because VCs can be simplified away, leaving "orphaned" lines of code (i.e., lines not referred to by any VC); note that simplification is in general necessary to ensure that all VCs can be proven by the ATPs In this case, however, the complexity of the domain meant that enough VCs were left.

Next we looked at optimized code, which was generated from a different, earlier model of the same system. The code size was larger because of inlining, amounting to 792 lines. We certified two frame requirements, yielding 64 proven VCs, and a document of 17 pages, evenly split between

both requirements. Note the similar size of the explanations per requirement: the optimization only affects code size, but not its conceptual complexity. The first requirement corresponded to 31 VCs and 126 LOC; 124 of these were covered by both by VCs and documentation. Here, VC simplification orphaned a local variable initialization, and the document did not cover a top-level conditional, because this did not correspond to any schema. For the second requirement we got very similar results.

7. Related Work

Early work on program comprehension recognized the cognitive difficulty in understanding concepts that are distributed throughout the code. *Plans* [17] and *focusing* [19] are approaches to recognizing such concepts and are similar to weak forms of our schemas, though they do not verify the selected fragments. The emphasis shifted later to more probabilistic approaches based on information retrieval [20] that seek to recover traceability links from informal artifacts. The earlier exact approaches, however, are more relevant to the problems we are addressing.

The various notions of concept have received considerable attention in program understanding [21]. In this work, we took concept to refer to the elements of a mathematical domain theory, such as quaternion, or angular velocity.

Antkiewicz et al. [1] use code queries, which are approximations to structural and behavioral patterns, in order to reverse engineer framework-specific models from framework code. It is similar to our work in the sense that we use patterns to reverse engineer "logical structure".

The problem of frame safety has been addressed by Lowry et al. [18], who used a domain-specific type system to verify the safety of abstract geometric calculations. The language analyzed was quite simple, however, so that annotations could be restricted to the declarations of the input variables, with no need for the inference of patterns or intermediate annotations. Although the underlying domain knowledge is similar to that used in our example, this is a very specific solution, in contrast to our "retargetable verifier".

Most of the previous work on proof documentation has focused on translating low-level formal proofs, in particular those given in natural deduction style. [2] presents an approach that uses a proof assistant to construct proof objects and then generates explanations in pseudo-natural language from these proof objects. However, this is based on a low-level proof even when a corresponding high-level proof was available. The Proverb system [12] renders machine-found natural deduction proofs in natural language using proof reconstruction. It defines inference rules for an intermediate representation called *assertion level* and abstracts machine-found resolution proofs using these rules; these abstracted proofs are then verbalized into natural language. Such an approach allows atomic justifications at a higher level of abstraction. In [11], the authors propose a new approach to text generation from formal proofs exploiting the high-level interactive features of a tactic-style theorem prover. It is argued that tactic steps correspond approximately to human inference steps. None of these techniques, though, is directly concerned with program verification. There has also been research on providing formal traceability between specifications and generated code. [29] presents a tool that indicates how statements in synthesized code relate to the initial problem specification and domain theory. In [31], the authors build on this to present a documentation generator and XML-based browser interface that generates an explanation for every executable statement in the synthesized program. It takes augmented proof structures and abstracts them to provide explanations of how the program has been synthesized from a specification.

The Whyline tool [14] supports *interrogative debugging*, where techniques from program analysis are used to answer user queries about program behavior. This is similar to our work in that verification techniques are used to provide explanations in terms of concepts from a domain.

8. 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 subsystem. We can also generate review documents for simpler execution-safety style properties.

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, domain concepts, and the relevant lines of code, as well as the generated verification conditions. It also provides *verifiably correct* tracing between requirements and assumptions, and consequently, can detect unused assumptions. Traditional approaches to tracing do not consider V&V artifacts. Future work would be to also check for inconsistent requirements and assumptions.

The tool is aimed at facilitating code reviews, 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. By verifying and providing insight into the code, it effectively does this for the original model as well.

In our preliminary evaluation we found that misleading identifiers can be a real problem, in particular when code is developed by multiple contractors; this experience was confirmed by domain experts. Addressing the reviewer's suggestion to present the explanations in a forwards manner

from assumptions to requirements would best be achieved by making use of more systematic techniques from NLG.

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.

It is clear that 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. Also, more information could be gleaned from the proofs, such as the use of constants and lookup tables. We also continue to extend the underlying domain theory that is used to verify the code. 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 [15] 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] Y. Coscoy, G. Kahn, and L. Thery. Extracting text from proofs. In *Proc. Second Intl. Confl. Typed Lambda Calculi and Applications*, *LNCS 902*, pp. 109–123. Springer, 1995.

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

[4] E. Denney and B. Fischer. Correctness of source-level safety policies. In *FM'03*, *LNCS 2805*, pp. 894–913. Springer, 2003.

[5] E. Denney and B. Fischer. Annotation inference for the safety certification of automatically generated code. In *ASE'06*, pp. 265–268. IEEE, 2006.

[6] 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.

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

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

[9] 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. IEEE, 2008.

[10] T. Erkkinen. Production code generation for safety-critical systems. Technical report, MathWorks, 2004.

[11] A. M. Holland-Minkley, R. Barzilay, and R. L. Constable. Verbalization of high-level formal proofs. In *AAAI/IAAI*, pp. 277–284, 1999.

[12] 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.

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

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

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

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

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

[18] M. Lowry, T. Pressburger, and G. Roşu. Certifying domain-specific policies. In *ASE'01*, pp. 118–125. IEEE, 2001.

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

[20] R. Oliveto. Traceability management meets information retrieval methods: Strengths and limitations. In *Proc. 12th European Conf. Software Maintenance and Reengineering*, pp. 302–305. IEEE, 2008.

[21] V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *Proc. 10th Intl. Workshop Program Comprehension*, pp. 271–278. IEEE, 2002.

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

[23] J. Richardson and J. Green Traceability through Automatic Program Generation In *Proc. 2nd Intl. Workshop Traceability in Emerging Forms of Software Engineering*. Montreal, 2003.

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

[25] S. Schulz, E — A Brainiac Theorem Prover, *AI Communications*, (15):111–126, 2002.

[26] I. Stürmer and M. Conrad. Test suite design for code generation tools. In *ASE'03*, pp. 286–290. IEEE, 2001.

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

[28] D. A. Vallado. *Fundamentals of Astrodynamics and Applications*, 2nd ed. Microcosm Press / Kluwer, 2001.

[29] J. Van Baalen, P. Robinson, M. Lowry, and T. Pressburger. Explaining synthesized software. In *ASE'98*, pp. 240–248. IEEE, 1998.

[30] C. Weidenbach, SPASS home page, www.spass-prover.org

[31] J. Whittle et al. Amphion/NAV: Deductive synthesis of state estimation software. In *ASE'01*, pp. 395–399. IEEE, 2001.