Arguing Conformance

Patrick Graydon, Ibrahim Habli, Richard Hawkins, and Tim Kelly, University of York

John Knight, University of Virginia

// Explicit and structured arguments can clarify and substantiate claims of conformance to software assurance standards and improve the process of conformance assessment. //

CONFORMANCE TO SOFTWARE assurance standards plays an essential role in high-integrity (particularly safety-critical) systems across many domains. Such standards can guide or constrain development processes, prescribe or proscribe product features, and dictate assessment practices. They help establish a consistent benchmark against which we can measure projects, define both minimum standards and best practices, and improve maturity in software development and assessment.

Transferring confidence between stakeholders in high-integrity software projects is essential. For example, airline customers want assurance that airborne software won't compromise their safety. However, neither self-reports of conformance nor independent confirmation of conformance is a perfect means to effect this transfer. Problems stem mainly from the need to interpret the text of a standard to fit the specifics of a particular application. Although documenting conformance is commonplace, the quality, transparency, and scrutability of documentation can vary significantly. This results in a lack of clarity as to exactly what a conformance claim signifies. We propose using explicit, rigorous, and structured conformance arguments to transfer confidence in software integrity.

Software Assurance Standards

Standards and their assessment mechanisms vary widely. Some, such as CAP 670 SW01, specify high-level goals, requiring applicants to provide "argument and evidence … which show that the software satisfies its safety requirements."[1] Others, such as IEC 61508, mandate low-level details: applicants must use "static synchronization of access to shared resources" or justify their failure to comply.[2] Some standards, such as RTCA DO-178B, specify some details of the assessment process.[3] Others—such as IEC 61508—don't, although applicants can choose to pay a third party to audit their conformance claims.

Transferring Confidence

Conformance to an assurance standard serves two main purposes in the development of high-integrity software. First, conformance can help *ensure* integrity by influencing software design and implementation. For example, IEC 61508 dictates the use of best practices in design and implementation.[2] Second, conformance can help establish *assurance* of integrity by influencing software assessment practice and guiding the acceptable forms of evidence. For example, DO-178B requires developers to collect test, analysis, and review reports as evidence that a system satisfies the standard's objectives.[3]

In many contexts, it's necessary to transfer confidence in a software system's integrity to stakeholders who aren't developers. In some cases, transferring confidence requires conveying detailed information about a system's specific properties. For example, suppose that a commercial aircraft reuses an operating system originally designed for automobiles. Aircraft designers and regulators must have confidence that the properties that made the operating system adequately safe for automobiles also make it adequately safe for commercial aircraft.

Confidence derived from the use of a software assurance standard is generally transferred via conformance or

0740-7459/12/$31.00 © 2012 IEEE

compliance mechanisms. An artifact *conforms* to a standard if it voluntarily meets the requirements of that standard. Transferring confidence in self-assessed conformance requires that the stakeholder trust the developers' claims of conformance. In contrast, an artifact *complies* if a regulator forces it to meet the requirements; this typically results in a certificate attesting to compliance. Transferring confidence through this mechanism requires that the stakeholder trust that the regulator's assessment established all the required properties. Neither conformance nor compliance mechanisms are wholly sufficient.

The Shortcomings of Confidence-Transfer Mechanisms

Standards requirements fall into four categories on the basis of whether they constrain

- the development process,
- internal product attributes,
- external product attributes, or
- resources used in production.[4]

Unlike standards in other industries or software interface and product standards, software assurance standards have a high proportion of process requirements. For example, the XML standard comprises mainly external product attribute requirements.[5] In contrast, IEC 61508 contains many requirements such as, "The compatibility of the tools of an integrated toolset shall be verified" (7.4.4.9) and "Programming languages for the development of all safety-related software shall be used according to a suitable programming language coding standard" (7.4.4.12).[2]

Determining conformance with such requirements isn't as simple as just testing whether a given XML document has the required structure. Despite calls to avoid these kinds of requirements,[4] current standards (including the recently updated IEC 61508) contain them. Moreover, it might be impossible to completely eliminate such requirements in software assurance standards.

Software assurance standards are meant to apply to broad classes of software. Standards authors aim to both ensure and assure integrity as much as is practicable without unnecessarily constraining the development process. As a result of these laudable goals, software assurance standards contain requirements that we must interpret in the context of each system; neither self-assessment of conformance nor independent assessment of compliance is always straightforward and unambiguous. Facilitating adequate transfer of confidence requires greater exposition and transparency than existing conformance and compliance practices provide.

Transfer of confidence requires greater exposition and transparency than existing conformance ... practices provide.

Standards and Interpretation

Standards authors are well advised to make requirements as clear and objectively verifiable as possible. Nonetheless, there are at least three distinct scenarios in which interpretation is still necessary:

- the use of high-level goals in the standard,
- deliberate nonspecificity in the standard, and
- the possibility of meeting the letter but not the spirit of the standard.

A few examples will help us illustrate this.

High-level goals. As an example of how high-level goals require interpretation, consider CAP 670 SW01's requirement that "software implemented as a result of software safety requirements is not interfered with by other software."[1] If safety requirements constrain execution time, we must interpret interference to include cache contention. If not, this interpretation is too broad.

Nonspecificity. Many assurance standards contain deliberate nonspecificity. For example, IEC 61508 requires the developer to "select and justify an integrated set of techniques and measures necessary during the software safety lifecycle phases to satisfy the Software Safety Requirements Specification."[2] To achieve this (in part), developers must use "cyclic behavior, with guaranteed maximum cycle time," a "time-triggered architecture," or an event-driven architecture "with guaranteed maximum response time," or else justify an alternative choice. Deliberate nonspecificity permits developers to accommodate both a wide variety of systems and the effects of other design and development choices. This flexibility, however, requires interpretation.

Spirit of the standard. As an example of how it's possible to meet the letter but not the spirit of a standard, consider IEC 61508's requirements that "source code shall ... be readable, understandable, and testable" and that "each module of software code shall be reviewed."

The standard's authors might have intended "review" to include both examining the code for readability and computing and analyzing appropriate metrics, with adequate reporting and follow-up. However, if the developers interpret "review" to include only issues of code correctness, fail to use appropriate techniques, or don't act with the required diligence, the software won't meet the objective.

The necessity of interpretation also raises the possibility of misinterpretation. Some standards require rationales or justifications that partially address this problem. For example, DO-178B requires developers to prepare a software accomplishment summary describing how the standards' requirements have been met.[3] If these rationales and justifications were fully detailed and carefully structured and presented, they might form partial or complete conformance arguments. However, standards don't generally require developers to present rationales or justifications in the form of rigorous arguments, and this isn't common practice.

Compliance Assessment

One way to transfer confidence despite the need for interpretation is for an independent party—either a regulator or a third-party company hired by the developers—to assess compliance. The independent party contributes an impartial viewpoint and can refuse certification if the developers' interpretation is inappropriate. Unfortunately, compliance assessment isn't a complete solution for three reasons:

- it's imperfect,
- it's costly, and
- it can obscure details needed for reuse.

If compliance assessment were perfect, it would be repeatable: different assessors would always return similar judgments for similar applications. Adequate processes, appropriately skilled and trained assessors, and accreditation of assessment services increase repeatability. However, anecdotes about developers shopping around for lenient assessors suggest that assessments aren't always perfectly repeatable in practice—we need greater transparency.

Independent compliance assessment is valuable. Unfortunately, it's also expensive. For example, the US Government Accountability Office found that Evaluation Assurance Level 4 Common Criteria evaluations take between nine and 24 months and cost between US$140,000 and $340,000.[6] Concerns about the cost of compliance assessment inhibit its use. For example, the US Federal Aviation Administration is replacing its system of assessment by designated engineering representatives with a system in which development organizations self-assess conformance. When conformance is self-assessed, transfer of confidence to stakeholders requires a reason to believe that the developers have interpreted the standard appropriately.

A conformance argument ... show[s] that an artifact adequately meets the standard's requirements.

Present compliance assessment practice can also be insufficient when developers attempt to reuse software in a different domain. Assessing software for cross-domain use requires knowing what specifically can be claimed of the software and its development process and having evidence substantiating those claims. A certificate of compliance is evidence that the system complies in some way with the text of each of the standard's requirements but it doesn't give the details of how the system complies.

Conformance Arguments

Our prior work with safety arguments suggested a new approach to transferring confidence: argue conformance explicitly. A safety argument is a structured argument that makes explicit the developers' rationale for the claim that a system is adequately safe to operate in a given context. The argument explains the available evidence, showing how it supports this conclusion. A conformance argument brings rigor and transparency to a different conclusion: not safety, but conformance. We define a conformance argument as a structured, comprehensive, and defensible argument demonstrating that the evidence is sufficient to show that an artifact adequately meets the standard's requirements.

Conformance arguments, like safety arguments, are rigorous, but necessarily informal, logical arguments. Each decomposes a main claim into a series of subclaims until these can be solved with evidence. The novelty of a conformance argument lies in using explicit and structured argumentation to solve the problem of transferring confidence.

The difference in focus between safety arguments and conformance arguments leads to different main claims and different structures. A safety argument documents how evidence supports a belief in system safety, and we can use it even where no safety standard prevails. Typical safety arguments decompose the main safety claim over system hazards, showing how each hazard has been sufficiently managed.

In contrast, a conformance argument justifies belief in conformance, even if there's no compelling reason to believe that conformance is adequate evidence of safety. As a result, the first level of decomposition in a conformance argument is over the standard's requirements. Claims that each requirement has been satisfied are further decomposed until each subclaim can be supported by evidence.

Notation

Conformance arguments can be recorded in any suitably expressive argument notation. Assurance arguments have been recorded in structured text, the graphical Goal Structuring Notation (GSN),[7] and the Claims-Argument-Evidence (CAE) notation.[8] Work continues on a unifying argumentation metamodel.[9] Support for argument construction includes a Visio plug-in for GSN, the commercial Adelard Safety Case Editor for CAE and GSN, and other tools.

Use Cases

Three use cases exist for a conformance argument:

- *Self-assessment*. Developers self-assess conformance. They prepare a conformance argument to help them both identify any nonconformities and question their interpretation of the standard.
- *Independent assessment, including argument*. Developers prepare a conformance argument and submit it and the conformance evidence to independent conformance assessors. The argument both conveys and justifies the developers' interpretation of the standard.
- *Preassessment check*. Developers prepare a conformance argument prior to independent conformance assessment but don't submit it. The argument serves to help them ensure that their interpretations are reasonable, their evidence sound, and their justifications defensible before the assessment begins.

In all three cases, developers can show the argument to potential users or customers as needed to transfer confidence. For example, returning to our earlier example of an automotive operating system reused in an aviation context, aerospace developers practicing self-assessing might use an argument showing the OS's conformance to an automotive safety standard to understand whether and how the OS might conform to DO-178B.

Also, in any of the three cases, the developers might create both a safety argument and a conformance argument. For example, a single product used in both the UK military and US civil aerospace contexts would require both a safety argument and conformance to DO-178B. In such cases, the safety and conformance arguments might overlap significantly. For example, a safety argument might reference a conformance subargument that shows that a system uses a fault-tolerant architecture as required by a standard to show mitigation of a particular hazard.

A conformance argument ... illustrates the developers' interpretation of the standard.

Conformance Arguments' Benefits

The chain of reasoning in a conformance argument both illustrates the developers' interpretation of the standard and defines what each item of evidence must show if a specific system is to conform to a given standard. This transparency facilitates greater confidence in a conformance claim, improved predictability and repeatability of assessment outcomes, and cross-domain reuse of high-integrity software.

Explicitness

Making a conformance argument explicit enables careful review—by developers, regulators, third parties, or some combination of these—which can find defects in its reasoning. Because these defects might hide instances of nonconformance, confidence in conformance can be raised by detecting and repairing them.

Consider again our example of software that must be readable and understandable to conform to IEC 61508. For a code review to satisfy this requirement,

- the review procedure would have to direct reviewers to consider these matters,
- adequately skilled and trained reviewers would have to follow it faithfully, and
- developers would have to redress unreadable or incomprehensible code flagged during the review.

A careful and thorough compliance assessor might raise questions about these details. However, once the compliance logic is documented explicitly, criticism of that logic easily raises these questions. Moreover, having addressed them in an argument submitted to assessors signals that the developers were aware of the issues.

Transparency

In traditional compliance assessment processes, assessors converse with developers to explore the developers' understanding of the standard and rationale for believing that the program met its requirements. In the independent assessment, including argument use cases, introducing a rigorous argument makes these ad hoc, undocumented conversations both transparent and explicit. The added structural rigor that comes from treating the compliance rationale as an argument enables greater criticism, whereas the written record increases the transparency of the process.

Confidence

Some might hypothesize that critical examination of the argument is less useful than a more detailed prescription for acceptable evidence. For example, either a standard or its prescribed conformance assessment procedure might specify inspection of a randomly selected subset of code as a means of demonstrating conformance with a readability requirement. The prescription might require the subset to cover a specified percentage of the code and inspectors to have certain qualifications. Extending the prescription in this way does reduce the scope for developers to meet the letter of the standard, but not its spirit. However, it does so at the cost of flexibility: alternative approaches would be precluded, even if they provide greater confidence. For example, to confirm that software exhibits a required behavior, a standard could require testing that achieves branch coverage computed in a specified manner. However, that requirement might preclude the use of languages for which the specified procedure is inappropriate. Moreover, the requirement codifies the assumption—perhaps incorrectly in some applications—that branch coverage is adequate.

Design Efficiency

Critics might say that a prescribed evidence approach might spare inspectors the difficulty of approaching novel reasoning in each application. We don't anticipate developers inventing new forms of evidence with each application. They have incentive to use known forms of evidence to reduce both cost and risk. The use of well-explored lines of reasoning can be encouraged through the publication of patterns of known approaches to arguing conformance with each standard or regulation. Pattern documentation carries information that's useful to both developers and assessors, including specifications of circumstances in which the pattern is indicated or contraindicated as well as notes about known pitfalls. Publishing a catalog of conformance-argument patterns related to each standard or regulation would allow an argument-based approach while retaining the advantages of prescribed evidence.

Compliance assessments should be both predictable and repeatable.

Predictability and Repeatability

Compliance assessments should be both predictable and repeatable. If assessment is unpredictable, developers unnecessarily risk rework. An unrepeatable assessment doesn't accurately measure the property of interest. Making and criticizing a conformance argument improves the predictability and repeatability of compliance assessment outcomes by facilitating developer self-check and bounding the scope for assessor judgment, thus letting us use precedent to guide judgment.

Every conformance rationale, implicit or explicit, includes appeals to engineering judgment. For example, if a standard calls for modular design, we must judge whether a design is sufficiently modular; if a standard calls for testing to have structural coverage adequate to demonstrate that software meets its requirements, we must judge whether a test plan meets that criterion. A more detailed prescription can reduce the scope of judgment, but only at the cost of flexibility. Conformance argumentation can help eliminate the variability associated with judgment by limiting its scope. It does this by letting the developer encode a very specific scope and context for each judgment. For example, a developer might claim that a test plan is adequate because it achieves Modified Condition/Decision Coverage (MC/DC) and includes test cases for boundary values. In contrast to this broad judgment, a conformance argument might decompose this claim across types of defects, claiming that the test plan is adequate to detect defects in the source code's branch and loop structure because it achieves MC/DC as measured at the source code level. The narrower scope of this judgment offers less room for disagreement and thus greater predictability and repeatability in the certification process.

An Example

In collaboration with industrial partners, we have constructed conformance arguments for a variety of aerospace, automotive, and naval applications. These experiences demonstrate the feasibility of constructing conformance arguments and highlight the transparency that they bring. Here, we present a simplified extract from one of our arguments to illustrate how conformance arguments transfer confidence in

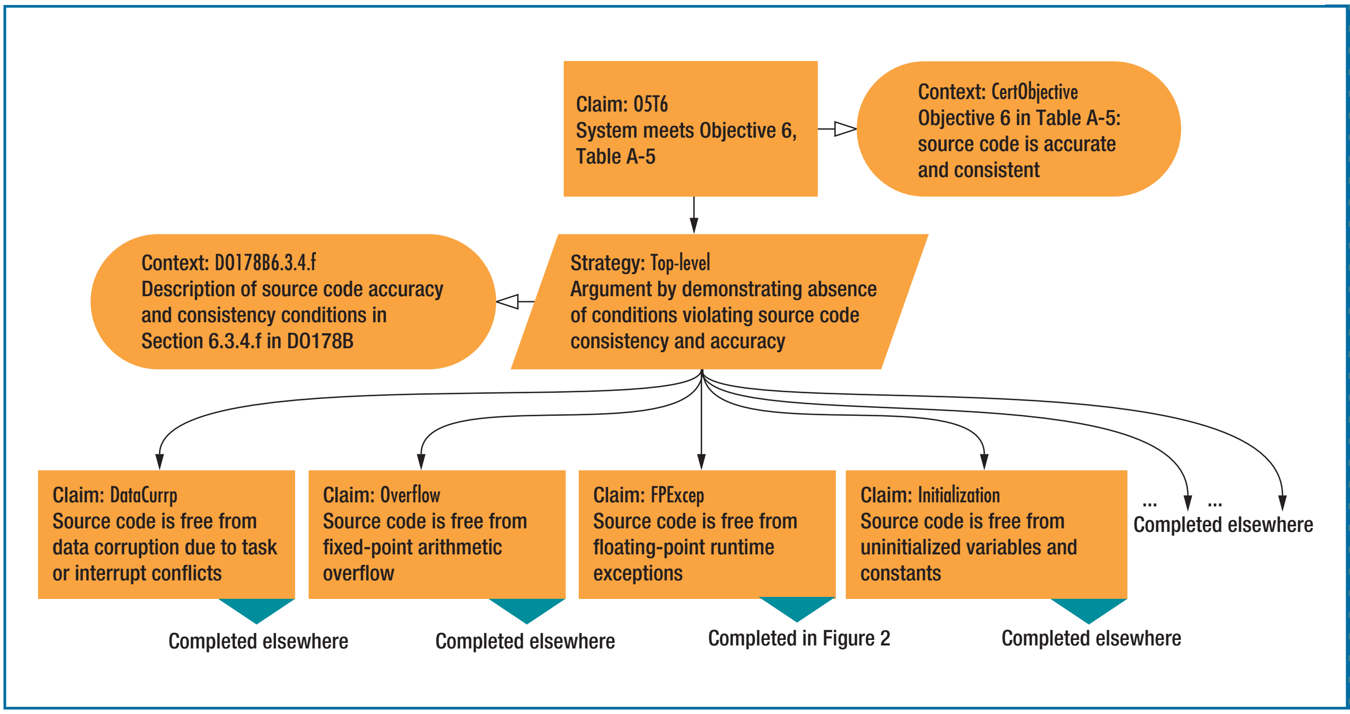
Claim: 05T6
System meets Objective 6, Table A-5
Context: CertObjective
Objective 6 in Table A-5: source code is accurate and consistent
Context: DO178B6.3.4.f
Description of source code accuracy and consistency conditions in Section 6.3.4.f in DO178B
Strategy: Top-level
Argument by demonstrating absence of conditions violating source code consistency and accuracy
Claim: DataCurrp
Source code is free from data corruption due to task or interrupt conflicts
Claim: Overflow
Source code is free from fixed-point arithmetic overflow
Claim: FPExcep
Source code is free from floating-point runtime exceptions
Claim: Initialization
Source code is free from uninitialized variables and constants
... ...
Completed elsewhere
Completed elsewhere
Completed elsewhere
Completed in Figure 2
Completed elsewhere

FIGURE 1. This conformance argument fragment shows how software for a commercial aircraft meets one of the objectives of DO-178B.

integrity. The example shows how software for a commercial aircraft meets one of the objectives of DO-178B.

Figure 1 shows part of a conformance argument represented in GSN. The top-most claim in the figure, 05T6, is a subclaim of the portion of the argument (not shown) that decomposes the main claim of DO-178B conformance over the standard's requirements (called *objectives*). Claim 05T6 is that objective 6 in Table A-5 of the standard, which requires that the source code be accurate and consistent, has been met.[3] The top-level strategy describes our approach: we argue over the different conditions that would violate source code consistency and accuracy. We include a reference to the standard's list of such conditions to justify this decomposition. For brevity, we elaborate on only one subclaim.

Figure 2 presents the rationale for believing that floating-point exceptions won't be raised at runtime. Rounded context element ExceptCause makes explicit our assumption that floating-point runtime exceptions are caused only by division by zero or by storing values that exceed type bounds. We support claim FPExcep by appealing to the use of a formal analysis technique, the Floating-Point Analysis Method (FPAM). The context element FPAM_Ref, associated with the strategy ExceptCause, summarizes the elements of FPAM and refers to the method's guide. Next are two claims concerning the freedom of the source code from conditions that could lead to floating-point runtime exceptions, as FPAM demonstrates. Figure 2 elaborates on the claim concerning the freedom of the source code from division by zero. This claim is eventually supported by two items of evidence: AN0803 and PR0804. AN0803 references documentation of the FPAM error model of possible errors in floating-point representation along with a rule base that defines how to process floating-point representations in terms of error bounds. PR0804 references three reports documenting test cases that demonstrate the ability of FPAM, using the rule base, to confirm absence of divide-by-zero problems where previously, pessimistically, code would have required further testing.

We presented the complete conformance argument from which this example was taken to auditors from certification authorities in the US and Europe. We intended for the argument to explain how the evidence FPAM generated can be used to meet a DO-178B objective. Our experience was encouraging: the regulators found that the arguments helped them understand the FPAM evidence.

Because standards must be interpreted in the context of each software system, the possibility of misinterpretation is unavoidable. Transferring confidence that the benefits of a standard have been realized from the developers to other stakeholders requires transparency. A structured conformance argument can provide this transparency, facilitating the transfer of confidence.

A conformance argument exposes the developers' interpretation of the standard to scrutiny. By doing so, conformance arguments facilitate greater confidence and should improve the predictability and repeatability of compliance assessment outcomes. Our experience with conformance arguments suggests that regulators find them useful as a means of understanding and criticizing the rationale for conformance claims.

Acknowledgments

We thank Jim Moore for sharing helpful information about current conformance mechanisms. US National Science Foundation grant 0716478 partially funded this work.

References

1. *CAP 670, Air Traffic Services Safety Requirements*, UK Civil Aviation Authority, 2010.
2. *61508-3, Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems — Part 3: Software Requirements*, Int'l Electrotechnical Commission, 2010.
3. *DO-178B, Software Considerations in Airborne Systems and Equipment Certification*, RTCA, 1992.
4. S.L. Pfleeger, N. Fenton, and N. Page, "Evaluating Software Engineering Standards," *Computer*, Sept. 1994, pp. 71–79.
5. T. Bray et al., eds., *Extensible Markup Language (XML) 1.0*, 5th ed., World Wide Web Consortium, 2008; www.w3.org/TR/2008/PER-xml-20080205.

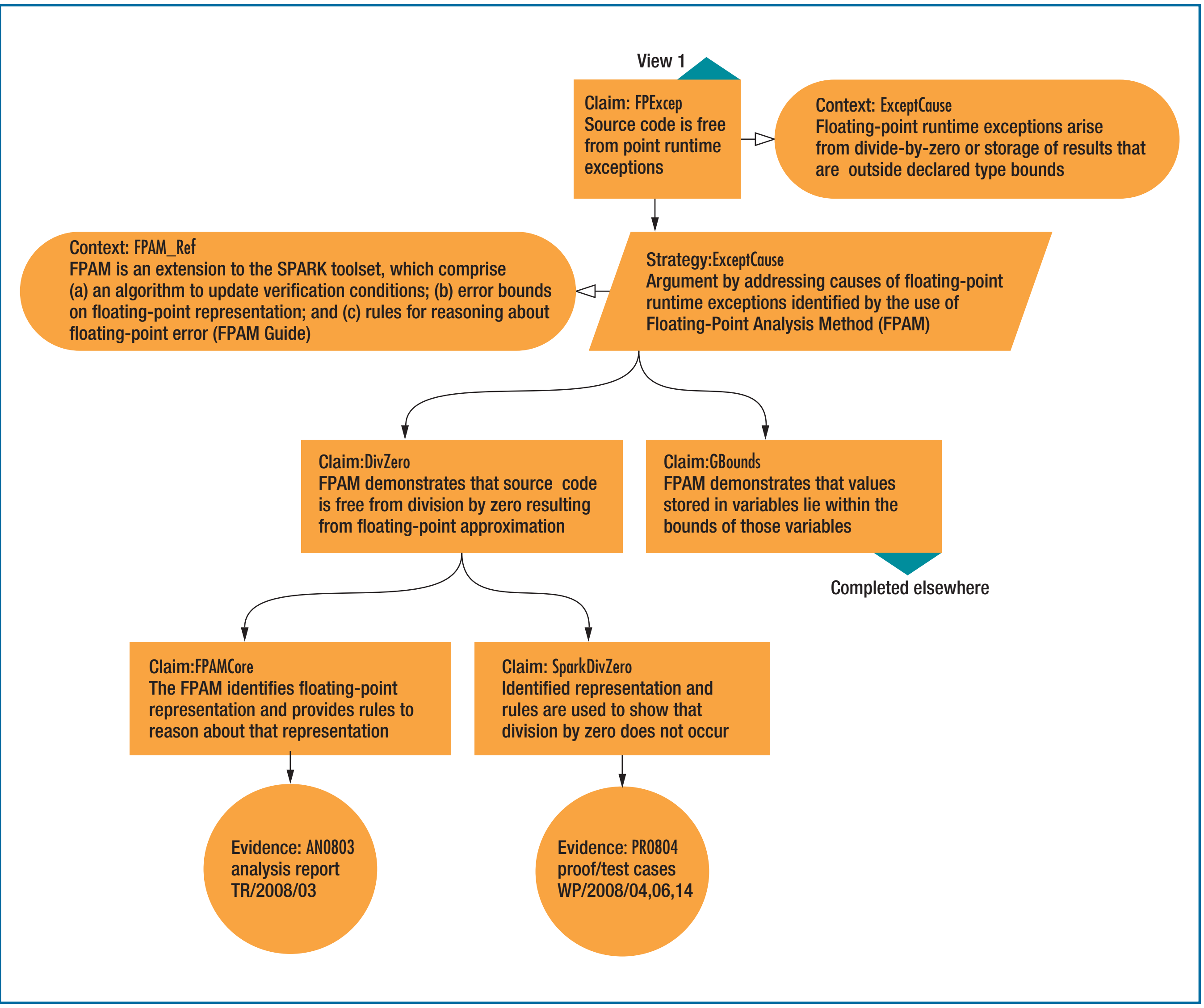
View 1
Claim: FPExcep
Source code is free from point runtime exceptions
Context: ExceptCause
Floating-point runtime exceptions arise from divide-by-zero or storage of results that are outside declared type bounds
Context: FPAM_Ref
FPAM is an extension to the SPARK toolset, which comprise (a) an algorithm to update verification conditions; (b) error bounds on floating-point representation; and (c) rules for reasoning about floating-point error (FPAM Guide)
Strategy:ExceptCause
Argument by addressing causes of floating-point runtime exceptions identified by the use of Floating-Point Analysis Method (FPAM)
Claim:DivZero
FPAM demonstrates that source code is free from division by zero resulting from floating-point approximation
Claim:GBounds
FPAM demonstrates that values stored in variables lie within the bounds of those variables
Completed elsewhere
Claim:FPAMCore
The FPAM identifies floating-point representation and provides rules to reason about that representation
Claim: SparkDivZero
Identified representation and rules are used to show that division by zero does not occur
Evidence: AN0803
analysis report
TR/2008/03
Evidence: PR0804
proof/test cases
WP/2008/04,06,14

FIGURE 2. An argument concerning freedom from floating-point runtime exceptions. Formal analysis evidence shows that the source code doesn't contain divide-by-zero or overflow errors.

6. *Information Assurance: National Partnership Offers Benefits, but Faces Considerable Challenges*, report GAO-06-392, US Government Accountability Office, 2006.
7. T. Kelly, "Arguing Safety—A Systematic Approach to Managing Safety Cases," doctoral dissertation, Dept. Computer Science, Univ. of York, 1998.
8. *ASCAD: The Adelard Safety Case Development Manual*, Adelard, 1998.
9. *Argumentation Metamodel (ARM)*, beta 1, Object Management Group, 2010; www.omg.org/spec/ARM/1.0/Beta1.

ABOUT THE AUTHORS

PATRICK GRAYDON is a research associate at the University of York. His research interests include safety and security argumentation, software engineering for certification, and certification processes. Graydon has a PhD in computer science from the University of Virginia. Contact him at patrick.graydon@cs.york.ac.uk.

IBRAHIM HABLI is a research and teaching Fellow in safety-critical systems at the University of York's Department of Computer Science. His research interests include software safety and certification, dependable architectures, and model-based development. Habli has a PhD in computer science from the University of York. Contact him at ibrahim.habli@cs.york.ac.uk.

RICHARD HAWKINS is a research associate at the University of York. His research interests include software assurance and safety case development. Hawkins has a PhD in computer science from the University of York. Contact him at richard.hawkins@cs.york.ac.uk.

TIM KELLY is a senior lecturer at the University of York. His research interests include safety case management, software safety analysis and justification, software architecture safety, certification of adaptive and learning systems, and the dependability of "systems of systems." Kelly has a DPhil in computer science from the University of York. Contact him at tim.kelly@cs.york.ac.uk.

JOHN KNIGHT is a professor of computer science at the University of Virginia. His research interests include software dependability and network security. Knight has a PhD in computer science from the University of Newcastle upon Tyne. He was the recipient of the IEEE Computer Society's 2006 Harlan D. Mills award and of the ACM SIGSOFT's 2008 Distinguished Service award. Contact him at knight@cs.virginia.edu.

IEEE Software

HOW TO REACH US

WRITERS
For detailed information on submitting articles, write for our Editorial Guidelines (software@computer.org) or access www.computer.org/software/author.htm.

LETTERS TO THE EDITOR
Send letters to

Editor, *IEEE Software*
10662 Los Vaqueros Circle
Los Alamitos, CA 90720
software@computer.org

Please provide an email address or daytime phone number with your letter.

ON THE WEB
www.computer.org/software

SUBSCRIBE
www.computer.org/software/subscribe

SUBSCRIPTION CHANGE OF ADDRESS
Send change-of-address requests for magazine subscriptions to address.change@ieee.org. Be sure to specify *IEEE Software.*

MEMBERSHIP CHANGE OF ADDRESS
Send change-of-address requests for IEEE and Computer Society membership to member.services@ieee.org.

MISSING OR DAMAGED COPIES
If you are missing an issue or you received a damaged copy, contact help@computer.org.

REPRINTS OF ARTICLES
For price information or to order reprints, send email to software@computer.org or fax +1 714 821 4010.

REPRINT PERMISSION
To obtain permission to reprint an article, contact the Intellectual Property Rights Office at copyrights@ieee.org.