NASA/TR-2004-212819

Product-oriented Software Certification Process for Software Synthesis

Stacy Nelson

February 2004

NASA/TR-2004-212819

Product-oriented Software Certification Process for Software Synthesis

Stacy Nelson
Ames Research Center, Moffett Field, California

National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California 94035-1000

February 2004

Product-oriented Software Certification Process for Software Synthesis

FOR

NASA Ames Research Center

Dated: September 25, 2003

Contributors (in alphabetical order by first name):

Stacy Nelson, Nelson Consulting (POC)
Bernd Fischer, RIACS
Ewen Denney, QSS
Johann Schumann, RIACS
Julian Richardson, RIACS
Phil Oh, RIACS

TABLE OF CONTENTS

RECORD OF REVISIONS

REVISION	DATE	SECTIONS INVOLVED	COMMENTS
Initial Delivery	8/15/03	All Sections	Draft for review
Update	9/25/03	Sections 2.5 and 2.6	Version 1.0 finished

1. EXECUTIVE SUMMARY

The purpose of this document is to propose a product-oriented software certification process to facilitate use of software synthesis and formal methods. Why is such a process needed? Currently, software is tested until deemed bug-free rather than proving that certain software properties exist. This approach has worked well in most cases, but unfortunately, deaths still occur due to software failure. Using formal methods (techniques from logic and discrete mathematics like set theory, automata theory and formal logic as opposed to continuous mathematics like calculus) and software synthesis, it is possible to reduce this risk by proving certain software properties. Additionally, software synthesis makes it possible to automate some phases of the traditional software development life cycle resulting in a more streamlined and accurate development process.

The product-oriented approach views software components like individual units that can be certified similar to the Underwriter's Laboratory (UL) certification for tangible products, like fireproof doors. To certify that a door is fireproof, UL provides specifications for building a fireproof door then tests the door to ensure it meets these specifications. If the door passes the test, then it is certified as fireproof. While this approach is similar to the traditional software development process, it differs in one important way: the door has been proven to be fireproof, not merely tested until it is believed to be fireproof. Properties of software components can be tested and certified in the same manner.

The NASA Automated Software Engineering (ASE) team has developed techniques and tools, called software synthesis, to check the software components using a process similar to Underwriter's Laboratory certification process. NASA ASE provides guidelines for building formal specifications. Then, using these formal specifications, special tools generate the software and attempt to prove, via mathematical means, that a property has been satisfied. If the software properties can be proven mathematically, then it is deemed to be certified and the safety analysis based on the mathematical proof becomes the certificate.

In order to explain the details of the proposed product-oriented certification process, this document contains the following sections:

- Introduction – contains the definition and rationale for product-oriented software certification including an overview of software synthesis. Also, discusses industry standards based on formal proofs and the benefits of the product-oriented approach versus a traditional process-oriented approach.
- Product-oriented life cycle – includes a review of the traditional software life cycle and description of the new product-oriented life cycle. The new product-oriented life cycle describes automation of some life cycle phases for software components meeting synthesis criteria. These automated techniques can also reduce development costs because they can replace an infinite amount of testing by proving that properties are correct.
- Product-oriented safety case – compares the traditional safety justification (called a "Safety Case") to a product-oriented safety justification
- Approval/Certification Process - reviews the rationale behind traditional Test Readiness Review (TRR), Flight Readiness Review (FRR) and Approval/Certification Decision making. Explains how product-oriented certification can streamline the decision making process and provide greater degree of safety assurance with lower development costs.
- Tools – describes tools to accomplish product-oriented certification including a description of software synthesis tools, comparison of synthesis to traditional V&V tools and a discussion of certification of synthesis tools.
- Artifacts - lists proposed artifacts and/or enhancements to traditional artifacts for product-oriented software certification, including automation of the production of the Software Design Document to improve accuracy of design while reducing development costs.

2. INTRODUCTION

Safety is a property of a system/software meaning that the system/software will not endanger human life or the environment. *Safety-critical* means that failure or design error could cause a risk to human life. Examples of safety-critical software can be found in nuclear reactors, automobiles, chemical plants, aircraft, spacecraft, et al.

Mission critical means the potential loss of capability leading to possible reduction in mission effectiveness[1] Examples of mission critical software can be found in unmanned space missions like Deep Space One and others.

Certification is the recognition by the certification authority that a software product complies with the requirements[2]. In the case of tangible items like a fireproof door, the Underwriter's Laboratory has a certification process whereby the door (having been built to a particular specification) is then torched (also of particular specification). If the door remains intact and doesn't burn, then it is certified as being fireproof.

Historically, software certification has been more complex and risky than certification of fireproof doors. Traditional methods involve review of artifacts to ascertain that the humans constructing the software have sufficient expertise and checks and balances to ensure safety. Generally, there is no proof that software is accurate, but rather an exhaustive (or as exhaustive as the budget allows) effort to find and remove anomalies (bugs).

For safety-critical software, a product-oriented approach like that used by UL would make it possible for software reviewers to rely upon proof that software is accurate rather than just trusting it is bug-free. However, until recently, this type of product-oriented certification was too time-consuming and expensive because it required highly skilled mathematicians. Fortunately, the NASA Automated Software Engineering group devised a way to automate these proofs. It is called software synthesis and automated analysis. Using software synthesis, it becomes increasingly more feasible to certify software components using a process similar to UL fireproof door certification.

The following section contains an overview of software synthesis. Subsequent sections describe nuclear power, defense and transportation industry standards supporting formal proofs. There is a discussion about the rationale for traditional certification techniques and an explanation of the Underwriter's Laboratory product-oriented certification technique. Finally, the benefits of the product-oriented approach over the traditional process-oriented approach are identified along with new software development artifacts and enhancements to traditional artifacts.

2.1. Certifiable Software Synthesis

Software synthesis is a technique for establishing that software is safe based on mathematical proofs. By adding special assertions to code, it is possible to analyze the code and produce a mathematical proof of correctness. This proof is then sent to a proof checker which says "yes" the proof is correct or "no" it is not. At NASA, this technique has been automated so that it occurs quickly and accurately. For purposes of certification, a certificate is a safety explanation of the mathematical proof such as the example in Appendix E.[3]

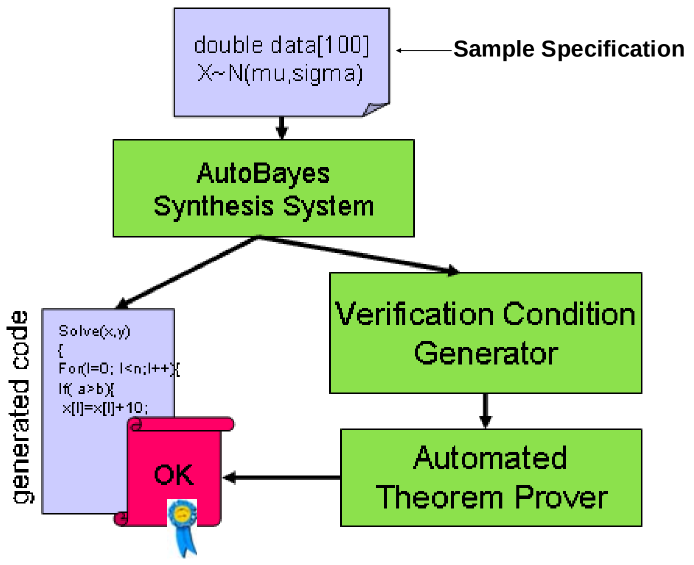


Figure 1: Software Synthesis

2.2. Other Standards Promoting Formal Proofs

The following standards support a product-oriented approach that relies upon formal methods including use of formal proofs:

- CE-1001-STD Rev. 1, Standard for Software Engineering of Safety-Critical Software, CANDU Computer Systems Engineering Centre for Excellence, January 1996 – used in Canada's Nuclear Power Industry
- DEF STAN 00-55, Requirements for Safety Related Software in Defence Equipment Part 1: Requirements and Part 2: Guidance, U.K. Ministry of Defence
- EN (European Norms) 50128:1997, Railway Applications: Software for Railway Control and Protection Systems, the European committee for Electrotechnical Standardisation (CENELEC)

2.2.1. Canadian Nuclear Power Industry: CE-1001-STD

Adopted in 1990, CE-1001-STD Rev. 1 focuses on three categories of safety systems in a nuclear power plant: shutdown systems, emergency coolant injection systems and nuclear generating containment systems. The proposed lifecycle includes: software development, verification and support processes (planning, configuration management and training). CE-1001-STD levies a minimum set of requirements on each lifecycle phase including specific quality objectives, quality attributes and fundamental principles that must apply to safety-critical software.

Primary quality objectives are safety, functionality, reliability, maintainability and reviewability. Secondary quality objectives are portability, usability and efficiency. The overall system, including software, must meet these quality objectives.

Quality attributes are also defined for safety-critical software including completeness, correctness, consistency, modifiability, modularity, predictability, robustness, structured, traceability, verifiability and understandability.

Fundamental principles include:

- Information hiding and partitioning – software design techniques in which the interface to each software module is designed to reveal as little as possible about the module's inner workings. This facilitates changing the function as necessary
- Use of formal methods – use of formal mathematical notation to specify system behavior and to verify or prove that the specification, design and code are correct and hence safe and reliable
- Specific reliability goals for safety-critical software
- Independence between development and verification teams

2.2.2. European Defence Industry: DEF STAN 00-55

DEF STAN 00-55 was written to capture the current best practices for developing and analyzing safety-related software. The standard defines the following key terms:

- Safety integrity - a measure of confidence that all safety features will function correctly as specified. The degree of safety integrity drives the design, development and assessment activities.
- Software Integrity Levels (SILs) – software is categorized based on Software Integrity Levels (SILs) that equate to the risk associated with the use of the software as follows:

    0. Non-safety related
    1. Low
    2. Medium
    3. High
    4. Very high

  Safety-critical software has SIL 4 and safety-related has SILs 1-4.

The life cycle for DEF STAN 00-55 consists of only six primary processes:

- Planning the system safety program
- Defining system safety requirements
- Performing a series of hazard analyses:
    - Functional analysis to identify hazards, associated with normal operations, degraded-mode operations, incorrect usage, inadvertent operation, absence of functions, and human error which causes functions to be activated too fast, too slow or in the wrong sequence
    - Zonal analysis to find hazards associated with usability on the part of the end users
    - Component Failure Analysis to find failure modes and rates of software components and the hardware where they operate
    - Operating and support hazard analysis to identify hazardous tasks which must be performed by end users and maintenance staff and ways to reduce potential for errors
- Allocating safety targets/requirements to system components

- Assessing achievement of safety targets
- Verifying the resultant systems safety is adequate and its individual and composite residual risk is acceptable. To accomplish safety integrity, DEF STAN 00-55 depends upon formal methods, formal specifications and formal proofs as part of the ongoing verification of completeness, consistency, correctness and unambiguousness of software engineering artifacts, particularly safety-related functions and features.

To assess risk, DEF STAN 00-55 defines:

- Four hazard severity categories (catastrophic, fatal, severe and minor)
- Six likelihood categories (frequent, probable, occasional, remote, improbable and implausible)
- Risk assessment matrix based on the hazard severity and likelihood with three levels (intolerable, undesirable and tolerable).

2.2.3. European Transportation Industry: EN 50128

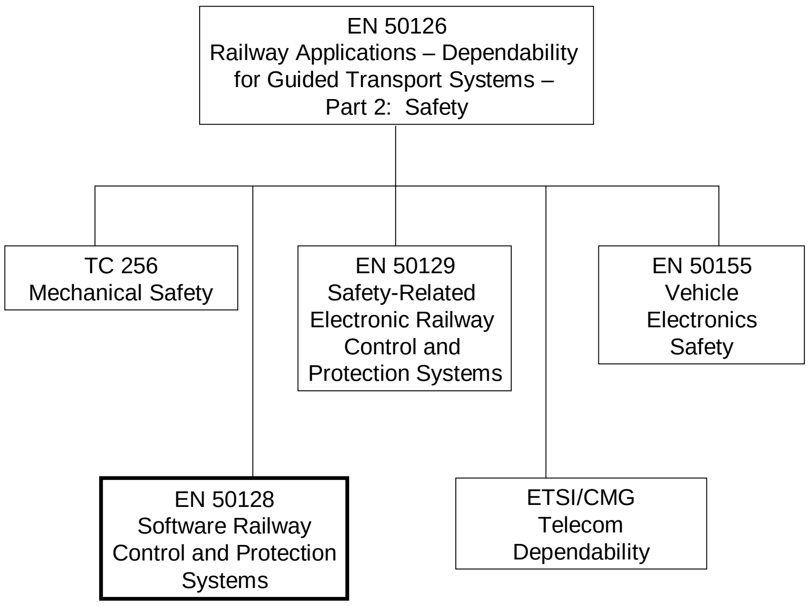

EN 50126
Railway Applications – Dependability for Guided Transport Systems – Part 2: Safety
TC 256
Mechanical Safety
EN 50129
Safety-Related Electronic Railway Control and Protection Systems
EN 50155
Vehicle Electronics Safety
EN 50128
Software Railway Control and Protection Systems
ETSI/CMG
Telecom Dependability

Figure 2: Structure of CENELEC Railway Dependability Standards

EN 50128 identifies *"methods which need to be used in order to provide software which meets the demands for safety and integrity".* It is organized around the concept of Software Integrity Levels (SILs) defined above.

All modules belong to the highest SIL unless partitioning can be demonstrated. Since SILs correspond to risk, EN 50126 defines a detailed risk classification scheme which utilizes a combination of qualitative and quantitative measures. EN 50126 defines six probability levels (incredible, improbable, remote, occasional, probable, frequent) and four safety hazard severity levels (catastrophic, critical, marginal, insignificant). It then correlates the hazard probability levels and safety hazard severity levels into four risk regions (intolerable, undesirable, tolerable and negligible). The standard provides directives for each region, for example: risk in the intolerable region shall be eliminated.

EN50128 defines seven lifecycle phases (requirements, specification, architecture specification, design and development, software/hardware integration, validation, assessment and maintenance). Two activities are ongoing throughout the lifecycle including: verification and quality assurance. Development begins only after system-level performance, safety, reliability and security requirements have been allocated to software.

EN 50128 also identifies activities, techniques and measures to be performed throughout the lifecycle based on the SIL to be achieved and assessed as shown in the table below. The following table contains techniques and measures by SIL for each lifecycle phase. Formal methods are recommended (R) for SIL 1-2 and highly recommended (HR) for SIL 3-4.

Table 1: EN 50128 Assignment of Techniques and Measures By SIL and Lifecycle Phase

Techniques and Measures	SIL 1-2	SIL 3-4	Lifecycle Phase
Structured methodologies	HR	HR	Requirements, Specification, Design and Development
Formal Methods (CCS, CSP, HOL, LOTOS, OBJ, Temporal Logic, VDM, Z)	**R**	**HR**	**Requirements, Specification, Design, Development and Verification**
AI, Dynamic Reconfiguration	NR	NR	Architecture Specification
Safety Bags, Recovery Blocks, Retry Fault Recovery	R	R	Architecture Specification
Partitioning, Defensive Programming, Fault Detection and Diagnosis, Error Detection, Failure Assertion, Diverse Programming, SFMECA, SFTA	R	HR	Architecture Specification
Design and coding standards Data Recording and Analysis	HR	M	Design, Development and Maintenance
Object-oriented Analysis and Design (OOAD)	R	R	Design and Development
Modular Approach	M	M	Design, Development
Static Analysis Dynamic Analysis	HR	HR	Verification
Software Quality Metrics	R	R	Verification
Functional Testing	HR	HR	SW/HW Integration and Validation
Probabilistic Testing Performance Testing	R	HR	SW/HW Integration and Validation
Modeling	R	R	Validation
Checklists Static Analysis Field Trials	HR	HR	Assessment

Techniques and Measures	SIL 1-2	SIL 3-4	Lifecycle Phase
Dynamic Analysis SFMECA, SFTA Common Cause Failure Analysis	R	HR	Assessment
Cause Consequence Diagrams Event Tree Analysis Markov Modeling Reliability Block Diagrams	R	R	Assessment
Change Impact Analysis	HR	M	Maintenance

M – Mandated, HR – Highly Recommended, R – Recommended, NR – Not Recommended, F - Forbidden

2.3. Traditional Process-Oriented Certification

Since safety-critical aerospace software is prevalent, what is the rationale behind certification of such software? In other words, how do engineers know when a new software product works properly and is safe to fly?

In the United States, software must undergo a certification process described in various standards by various regulatory bodies including NASA and the Requirements and Technical Concepts for Aviation (RTCA) which is enforced by the Federal Aviation Administration (FAA). While each NASA center and the FAA have unique certification processes, they share the same idea. Regulatory authorities will be looking for evidence that all potential hazards have been identified and that appropriate steps have been taken to deal with them. Europe has similar certification processes.

2.4. Safety Case

In order to meet current regulatory guidelines in the aerospace industry, developers construct a safety case as a means of documenting the safety justification of a system. The safety case is a record of all safety activities associated with a system throughout its life. Items contained in a safety case include the following:

- Description of the system/software
- Evidence of competence of personnel involved in development of safety-critical software and any safety activity
- Specification of safety requirements
- Results of hazard and risk analysis
- Details of risk reduction techniques employed
- Results of design analysis showing that the system design meets all required safety targets
- Verification and validation strategy
- Results of all verification and validation activities
- Records of safety reviews
- Records of any incidents which occur throughout the life of the system

- Records of all changes to the system and justification of its continued safety

The traditional safety case hinges on the expertise of key personnel and their ability to document risk reduction techniques and the historical reliability of the software. It does not generally include proof that software is safe because in the past, it would have been too time consuming for mathematicians to develop this proof manually.

For more information about traditional certification processes, see NASA/CR--2003-212806 *Certification Processes for Safety-Critical and Mission-Critical Aerospace Software* which contains:

- Standards For Safety-Critical Aerospace Software – lists and describes current standards including NASA standards founded upon IEEE/EIA 12207, as well as RTCA DO-178B
- Class A Versus Class B Software – explains the difference between two important classes of software: Class A dealing with safety-critical software and Class B for mission critical software
- DO-178BClass A Certification Requirements – describes special processes and methods required to obtain Class A certification for aerospace software flying under auspices of the FAA
- Dryden Flight Research Center (DFRC) Certification Process – documents the certification process used at Dryden for new aerospace software like the Intelligent Flight Control System including neural networks that adapt to flight conditions
- Jet Propulsion Lab (JPL) Approval Process – describes the approval process used at JPL for new space software like the Mars Smart Lander

2.5. Underwriter’s Laboratory (UL) Product-Oriented Approach

Underwriter’s Laboratory (UL) primarily focuses on safety certification of third party products because UL has no vested interest in the product; and therefore, can provide a more accurate assessment of safety for that product. Similarly, the Automated Software Engineering Group at Ames Research Center under the leadership of Dr. Michael Lowry can provide an assessment of software safety by leveraging certifiable software synthesis techniques.

UL testing procedures are based on standards. For example, UL 10A (Tin-Clad Fire Doors) and UL 10B (Fire Tests of Door Assemblies) are standards for fireproof doors. UL 10A contains the specification and is analogous to a software requirements specification (SRS) document. UL 10B describes test procedures and compares to the Verification and Validation Plan for software.

Upon completion of tests by UL, third parties are provided results via a formal test report much like the report provided by a software review board. However, when a product is certified by UL, the UL marking may be affixed showing consumers that the product has a safety certification. Currently, there is no such stamp of approval for software because the current process-oriented approach to certifying software cannot provide that level of assurance. At best, review boards for software approve deployment based on their belief that the development team has followed a sound process and that software demonstrations of key scenarios indicate that software is safe for all scenarios. Clearly, a product-oriented approach to software resulting in a certificate like UL would provide much more confidence that software is safe and reliable.

2.6. Product-oriented versus Process-oriented Approach

There are advantages and challenges to both the product-oriented and process-oriented approach to certification of safety-critical software. The advantages of the product-oriented approach for certification include:

- Mathematical proof that software components are accurate

- Easy third party (UL type process) certification process

Challenges revolve around scalability of this approach. Currently, it is only available at the software component level.

The process-oriented approach has the advantage of a well-established track record for success; however, it relies on human engineering and documentation which is time consuming and, unfortunately, software failures still result in loss of human life. It also limits the type of software that can be constructed. For example, neural networks have been developed that can be embedded into flight control systems making it possible for pilots to safely fly and land damaged aircraft. However, due to the complexity of neural networks, testing via traditional methods is generally not feasible.

In order to capitalize on the advantages of both approaches, this proposal describes a hybrid approach which combines key strengths from both the product and process-oriented approaches. The hybrid approach makes possible a smooth transition to new product-oriented techniques. It provides an avenue for the product-oriented approach to establish a track record for success (defined as better, safer, cheaper software).

The hybrid approach proposes automating part of the lifecycle resulting in a lifecycle much like the nuclear power, defense and transportation standards already relying upon formal proofs. Software meeting the synthesis criteria described below can be synthesized and other software can be developed via a traditional approach.

Synthesis Criteria

Synthesis is most cost effective when code is generated for a product family with sufficient variation that static libraries would not suffice.

Figure 3: Synthesis Criteria

Kalman filters are an example of code meeting synthesis criteria because Kalman filters are widely used in aerospace and other applications to reduce noise from sources like sensors. Additionally, Kalman filters are too complex to be coded into a static library. There are also software properties that lend themselves well to certification using annotations that can be generated as part of code synthesis process. Examples of these properties include: array safety (checking that array boundaries cannot be exceeded) and safety of mathematical functions (ensuring that divide by zero errors do not occur).

It is important to distinguish between software functionality and software safety features. Traditionally, software has been verified for functional correctness; however certifiable software synthesis makes it possible to check safety correctness properties to make sure the software won’t crash (e.g. array go out of bounds).

As confidence is cultivated for the product-oriented approach and more advanced synthesis techniques are developed, the hybrid approach can be expanded. Software synthesis can be applied to unique software components then to software integration, software architecture, etc. It can ultimately ensure the accuracy of the entire system.

3. PRODUCT-ORIENTED LIFE CYCLE

This section provides an overview of a typical life cycle then discusses changes necessary for a product-oriented life cycle.

3.1. Typical Process-Oriented Life Cycle

The following diagram shows the relationship between V&V and a typical, traditional life cycle described in current software development standards.

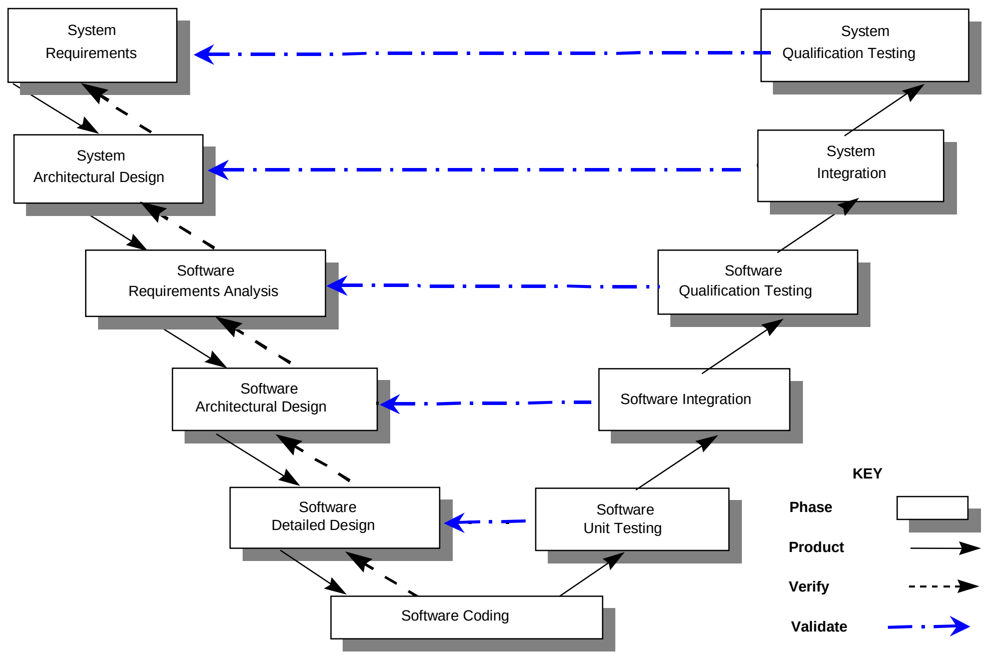


Figure 4: Typical Life Cycle Phases

A traditional system development project begins at the upper left of the diagram with system requirements, the top-level description of the operation of the system. V&V also begins at the inception of the development project as shown by the arrow pointing from system qualification testing (upper right) to system requirements. First, an overall V&V plan is developed. Initial V&V activities strive to ensure that system requirements can be tested. As the project matures, V&V is performed at each phase as shown by the arrows between phases. The following sections provide an overview of each life cycle phase.

3.1.1. System Requirements

Traditional system requirements must be stated in natural language in clear, concrete terms so they can be tested. An example requirement might be: software must execute at approximately 40Hz on an ARTS II computer. When writing requirements special consideration should be given to the following areas:

- Hardware Specifications: Description of hardware needs including CPU size and speed, number of CPUs (as may be used in a multi-processing), available on-board memory, interfaces between multiple cards in a system for data throughput, and any possible considerations for future extensions.
- Operating Capabilities: Description of how the product behaves. Requirements should include the operating frequency, allowed memory usage and desired computational precision. Modes of operation should also be discussed.

- Operating Environment: environment where the system will operate including the target performance envelope
- Algorithmic Capabilities: These include definitions of algorithms and pre-analyzed failure conditions.
- Data Recording Capabilities: Description of the kind of data that needs to be recorded including input data, output data, and data internal to the operation of the product
- Human User Interfaces: Description of what is needed so humans can use the system effectively

3.1.2. System Architectural Design

The system architectural design establishes a high-level software and hardware design based upon the system requirements. This begins the separation of the system requirements by function into system modules or sub-systems and establishes the means of data and control communication between the modules. The system architectural design should include a description of what the product does, what data is to be processed, and how it is interfaced with other systems/subsystems.

3.1.3. Software Requirements Analysis

Software requirements analysis is necessary to ensure the software requirements are based on the system requirements. Special software requirements for the product might include:

- Algorithmic Capabilities: More detailed requirements describing the product algorithms
- Hardware Specifications: requirements including allotted memory usage, allotted processor usage and perhaps constraints on specific execution times
- Operating Capabilities: explanation of operating capabilities and the different modes of operation
- Inputs/Outputs: Inputs and outputs to the product should be identified, as well as data recording capabilities. Software requirements add refinement to system requirements stating specifically:
    - Which data was being recorded
    - Precision of the recorded data
    - Frequency with which this data is recorded
    - Order this data is recorded
    - File format the data is recorded in and a description of how data is written to a file (one continuous file, sequences of files, single file which is always written to but never appended…)
- Human User Interfaces: Description of any human interface

3.1.4. Software Architectural Design

The software architectural design decomposes the high-level software design and describes software components and constructs needed to satisfy the software requirements.

3.1.5. Software Detailed Design

The software detailed design explains the details of how the software requirements will be satisfied. It should include a description of precise code constructs required to implement the product.

3.1.6. Software Coding

The software coding stage should include the actual product code. In some situations, a product may be implemented in a modeling language such as MATLAB/Simulink or Matrix-X and is auto-coded from the model into a desired programming language. In these situations, the software code would include the original system models.

3.1.7. Software Unit Testing

Software unit testing should include both black and white box testing.

3.1.8. Software Integration

Software integration should verify that the product as a whole works properly.

3.1.9. Software Qualification

Software Qualification Testing should ensure that the software requirements are sufficiently detailed to adequately and accurately describe the product.

3.1.10. System Integration

System integration testing should verify that the architectural design is detailed enough so, when implemented, the product can interface with system hardware and software in various fidelity testbeds.

System integration involves testing the system after it has been integrated into a larger system. This can include integration with target hardware and/or integration with onboard system computers and external pieces of software. Test results should identify successful completion of integration tests and any discrepancies or anomalies from expected outputs.

3.1.11. System Qualification Testing

System qualification testing should verify that the system requirements are sufficiently detailed to adequately and accurately describe the product to ensure that, when implemented, it will interface properly with the system in production.

3.2. Proposed Product-Oriented Life Cycle

In order to understand the product-oriented approach, consider a best-case scenario where all software requirements can be stated in formal terms resulting in automatic generation and certification of all code. The following diagram shows this theoretical approach.

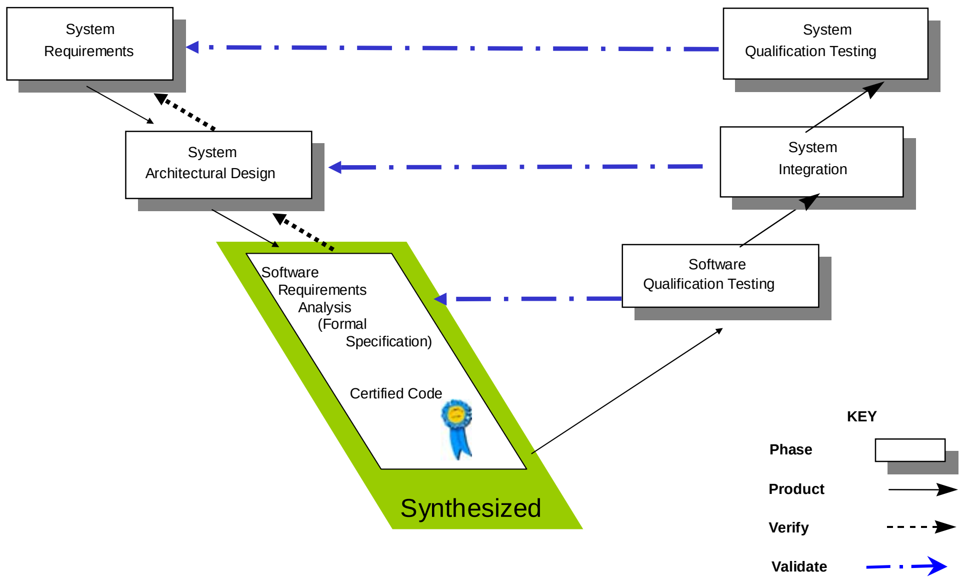

System Requirements
System Qualification Testing
System Architectural Design
System Integration
Software Requirements Analysis (Formal Specification)
Certified Code
Software Qualification Testing
Synthesized
KEY
Phase
Product
Verify
Validate

Figure 5: Theoretical Product-Oriented Approach

The theoretical approach begins like the traditional system development project at the upper left of the diagram with system requirements. V&V also begins like a typical system development project with system qualification testing (upper right). The two approaches remain identical until the software requirements analysis phase. At this point, a formal software specification is developed and validated. Then, code is automatically generated and certified. Finally, the certified code is integrated with the system and validated.

A formal specification is a special notation based on formal logic used to describe the properties of the software. When software is defined using this mathematical notation, code can be generated automatically along with one or more mathematical proofs to ensure the safety of the software. Proving properties increases the confidence in code, thereby reducing the need to perform manual V&V, making product-oriented software much less expensive and time consuming.

This type of theoretical approach will someday be a panacea for safety-critical systems – proven safety and very low development costs. The NASA Automated Software Engineering group is making great strides towards this panacea by making it possible to synthesize some software components. By integrating synthesized code with traditional code, it is possible to improve safety and reduce cost. The following diagram shows a hybrid approach where software meeting synthesis criteria are synthesized and unique software components are built using the traditional approach.

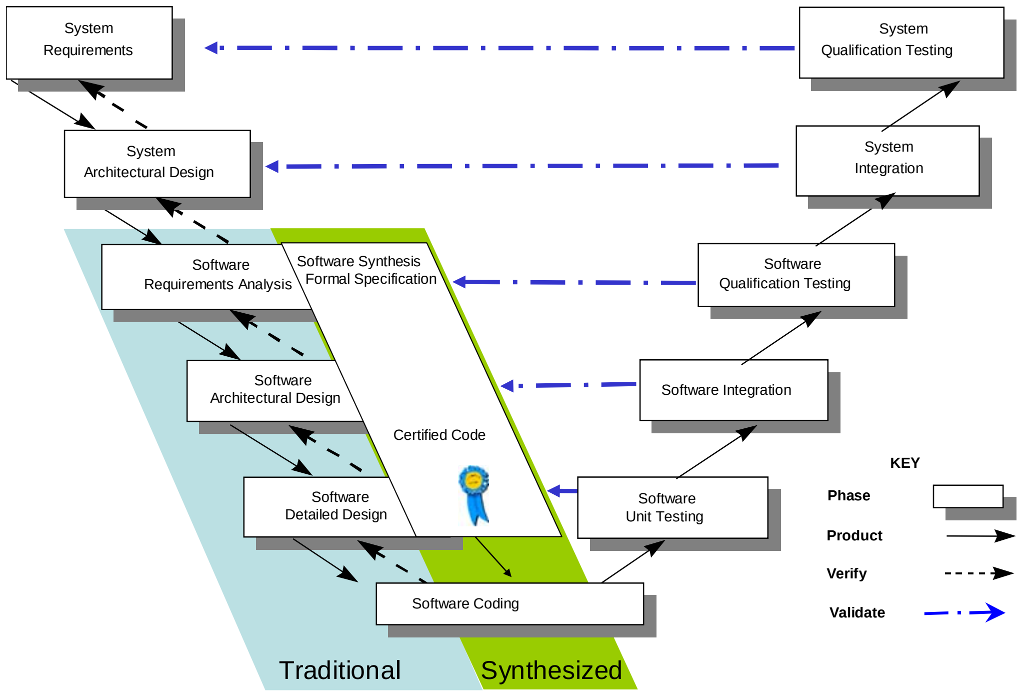

System Requirements
System Qualification Testing
System Architectural Design
System Integration
Software Requirements Analysis
Software Synthesis Formal Specification
Software Qualification Testing
Software Architectural Design
Software Integration
Certified Code
Software Detailed Design
Software Unit Testing
Software Coding
Traditional
Synthesized
KEY
Phase
Product
Verify
Validate

Figure 6: Hybrid Approach

Again, the first steps of this approach are identical to traditional software development. However, during software requirements analysis, software meeting synthesis criteria are developed using software synthesis, and unique components are built traditionally. Both types of components are integrated in the software integration phase and traditional V&V is performed.

Specific changes to life cycle phases include the following:

- Software requirements analysis – develop a formal specification and verify that it is correct *Note: experiments at IBM reveal that developing a formal specification for the entire system rather than just the components to be synthesized results in a more accurate system because of the analysis required for a formal specification. More accurate requirements generally lead to lower downstream development costs because less time is spent solving problems resulting from missing, conflicting or incorrect requirements.*
- Software architecture design – modularize software into software meeting synthesis criteria and unique components to take advantage of software synthesis and to maximize reusability
- Software Detailed Design – incorporate automatically generated SDD into software design document. Appendix D contains recommendations for incorporating the SDD into projects governed by MIL STD 498 and IEEE 12207.

4. PRODUCT-ORIENTED SAFETY CASE

The safety case is a record of all safety activities associated with a system throughout its life. The following tables lists items contained in a traditional safety case compared to items needed for a product-oriented safety case. Differences are highlighted in bold.

Traditional Safety Case	Product-oriented Safety Case
Description of the system/software	Description of the system/software
Evidence of competence of personnel involved in development of safety-critical software and any safety activity	Evidence of competence of personnel involved in development of safety-critical software and any safety activity
Specification of safety requirements	Specification of safety requirements **including formal specifications**
Results of hazard and risk analysis	Results of hazard and risk analysis
Details of risk reduction techniques employed	Details of risk reduction techniques employed
Results of design analysis showing that the system design meets all required safety targets	Results of design analysis showing that the system design meets all required safety targets
Verification and validation strategy	Verification and validation strategy **including software synthesis**
Results of all verification and validation activities	Results of all verification and validation activities **including certificates (safety explanation such as example in Appendix C)**
Records of safety reviews	Records of safety reviews
Records of any incidents which occur throughout the life of the system	Records of any incidents which occur throughout the life of the system
Records of all changes to the system and justification of its continued safety	Records **including certificates** for all changes to the system and justification of its continued safety

5. APPROVAL PROCESS

The approval process for traditional certification for safety-critical software is generally the same in military and commercial standards and across industries: the software must pass a series of reviews and be deemed safe enough to deploy. For example, the approval process used at NASA for safety-critical aerospace software is shown in the diagram below. Programs at NASA using this approach include the adaptive Intelligent Flight Control System (IFCS), the Mars Science Laboratory (previously, Mars Smart Lander, a rover to explore Mars) and Deep Space One experimental spacecraft.

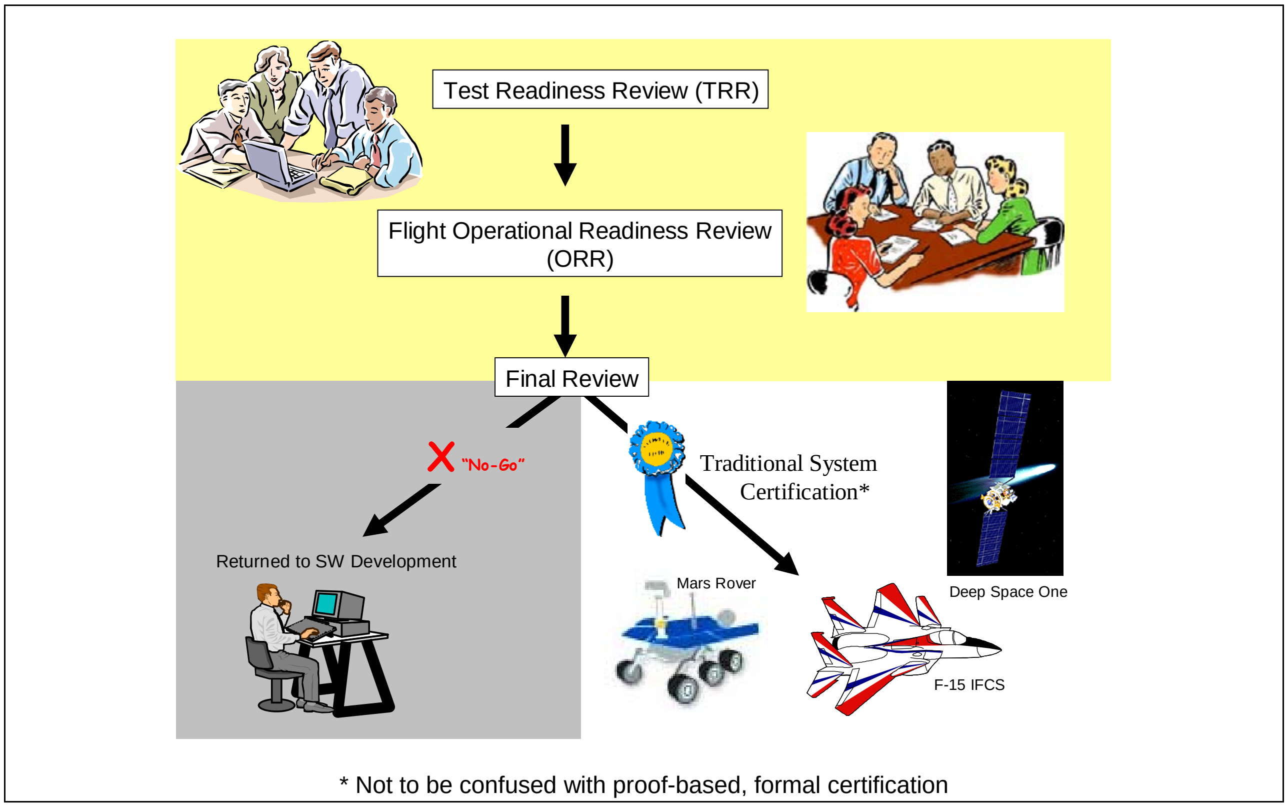

Test Readiness Review (TRR)
Flight Operational Readiness Review (ORR)
Final Review
"No-Go"
Returned to SW Development
Traditional System Certification*
Mars Rover
Deep Space One
F-15 IFCS
* Not to be confused with proof-based, formal certification

Figure 7: Traditional Certification Process

When systems and software are ready for approval they are reviewed by the internal project team at the Test Readiness Review (TRR). Once the software passes this internal review, it is reviewed by an independent team of engineers who have not worked on the project. The independent team conducts a Flight Operational Readiness Review (ORR). When the system and software pass the ORR, the Program Manager is notified and submits the project plans and preparations to the Final Reviewer(s). The Final Reviewer(s) determines whether the software is approved for implementation or must return to software development for further work.

In order for aerospace software to be implemented on commercial aircraft, it must be approved and certified by the FAA. The FAA certification process follows DO-178B and works in much the same way as the NASA approval process with the following exceptions:

- Technically, software is not certified, only aircraft components (flight control system, flight management system…) are certified. However, there are rigorous guidelines to ensure the software aspect of these components is safe, so we tend to speak about "software certification" even though software is not certified, per se.

- Negotiation is required to ensure optimal use of limited FAA review funds. A Plan for Software Aspects of Certification (PSAC) must be written describing who, what, when, where and how the software will be certified.
- More stringent requirements for up-to-date documentation. The FAA prefers to see documentation updated with final software features rather than the initial documentation with change memos attached. This reduces the complexity of reading the document; thus saving time for the reviewers. It also makes the documentation easier to understand and use. For example, the wiring diagrams of the Space Shuttle need to be rewritten because they contain an original document plus multiple revisions. Workers must flip through pages of revisions to find the latest wiring configuration. This is a slow, error-prone task.

The following diagram details the FAA approval process.

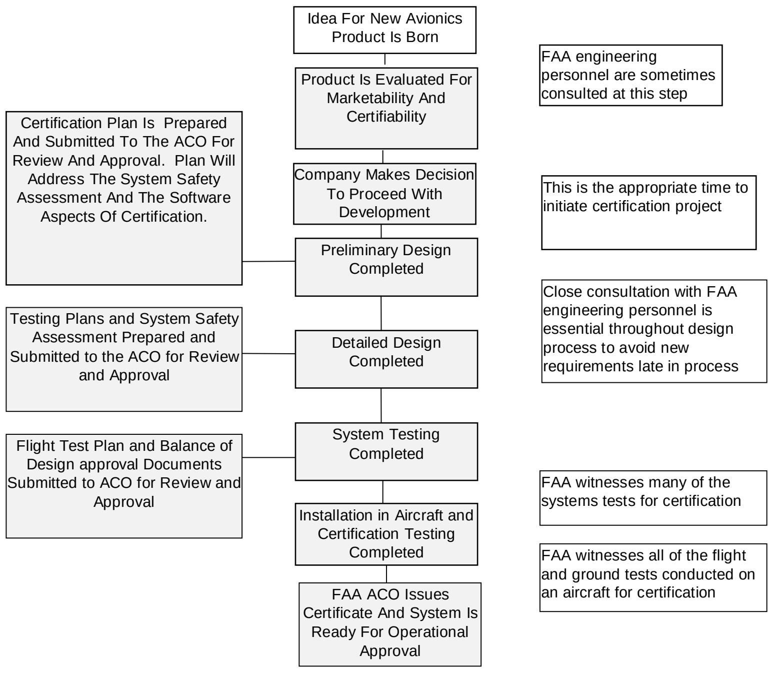

Idea For New Avionics Product Is Born
Product Is Evaluated For Marketability And Certifiability
FAA engineering personnel are sometimes consulted at this step
Certification Plan Is Prepared And Submitted To The ACO For Review And Approval. Plan Will Address The System Safety Assessment And The Software Aspects Of Certification.
Company Makes Decision To Proceed With Development
This is the appropriate time to initiate certification project
Preliminary Design Completed
Close consultation with FAA engineering personnel is essential throughout design process to avoid new requirements late in process
Testing Plans and System Safety Assessment Prepared and Submitted to the ACO for Review and Approval
Detailed Design Completed
Flight Test Plan and Balance of Design approval Documents Submitted to ACO for Review and Approval
System Testing Completed
FAA witnesses many of the systems tests for certification
Installation in Aircraft and Certification Testing Completed
FAA witnesses all of the flight and ground tests conducted on an aircraft for certification
FAA ACO Issues Certificate And System Is Ready For Operational Approval

Figure 8: FAA Approval Process

The approval process for the product-oriented approach remains the same as the traditional approach except that when software developers present certificates proving software correctness to the review boards, the reviewers will have a much greater degree of confidence that the software is safe. This increased confidence can reduce time spent in review.

6. TOOLS

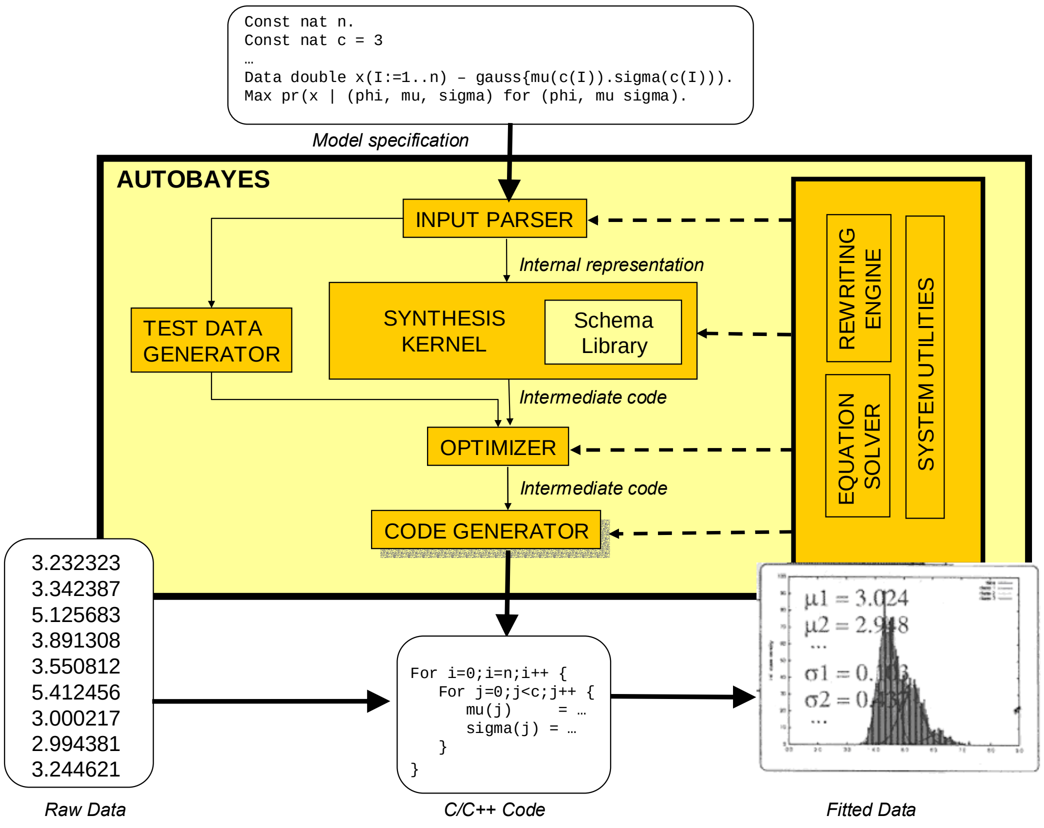
This section contains three sections:

- Description of synthesis tool used for Certifiable Software Synthesis at Ames Research Center
- Comparison of the synthesis tool to traditional V&V tools
- Discussion of synthesis tool certification

Automated program synthesis aims at automatically constructing executable programs from high-level specifications. It is usually based on mathematical logic, although a variety of different approaches exist[4]. Here, we will focus on a specific approach, schema-based program synthesis, and a specific system, AUTOBAYES.

Externally, AUTOBAYES looks like a compiler: it takes an abstract problem specification and translates it into executable code. Internally, however, it is quite different. AUTOBAYES[5] generates complex data analysis programs from compact specifications in the form of statistical models. It has been applied to a number of domains, including clustering, change detection, sensor modeling, and software reliability modeling, and has been used to generate programs with up to 1500 lines of C++ code.

First, AUTOBAYES derives a customized *algorithm* implementing the model and then it produces an optimized, imperative code implementing the algorithm. The following figure shows the system architecture:



```
Const nat n.
Const nat c = 3
…
Data double x(I:=1..n) – gauss{mu(c(I)).sigma(c(I))).
Max pr(x | (phi, mu, sigma) for (phi, mu sigma).
```

```
3.232323
3.342387
5.125683
3.891308
3.550812
5.412456
3.000217
2.994381
3.244621
```

```
For i=0;i=n;i++ {
   For j=0;j<c;j++ {
      mu(j)     = …
      sigma(j) = …
   }
}
```

Figure 9: AUTOBAYES System Architecture[6]

In the first processing step, the given specification is parsed and converted into internal format and a Bayesian network representing the model is constructed. Then the synthesis kernel analyzes the network, tries to solve the given optimization task, and instantiates appropriate algorithm schemas from a schema library. The schemas encode the domain knowledge. They contain rules to decompose the network into independent parts, rules to search symbolically for closed-form solutions and algorithm skeletons which are instantiated during the synthesis process. These schemas are guarded by applicability conditions. A schema consists of a program fragment with open slots and a set of applicability conditions. The slots are filled in with code pieces by the synthesis engine calling schemas recursively. The conditions constrain how the slots can be filled; they must be proven to hold in the given specification before the schema can be applied. Some of the schemas contain calls to symbolic equation solvers, others contain entire skeletons of statistical or numerical algorithms. By recursively invoking schemas and composing the resulting code fragments, AUTOBAYES is able to automatically synthesize programs of considerable size and internal complexity.

Figure 9 below shows in stylized Prolog-notation a slightly simplified schema which is selected when a function needs to be maximized. It synthesizes a code fragment C which calculates the maximum w.r.t. a single variable X for a symbolically given function F. The applicability of this schema is restricted to cases where a first derivative of F exists. The schema first tries to compute the maximum symbolically by solving the equation ∂F/∂X = 0 for X. If that succeeds, it returns a single assignment. Otherwise, an iterative numerical optimization routine must be synthesized in order to solve the given problem. Such an algorithm consists of three code segments: finding a start value *x*0, calculation of the search direction *p*, and the step length *λ*. Then, starting with *x*0, the maximum is sought by iteratively approaching the maximum: *xk*+1 = *xk* + *λkpk* (for details see [9, 17]). Our schema assembles this algorithm by recursive calls to schemas to obtain code fragments CInit, CSteplength, and CStepdir for initialization, calculation of the step length, and step direction, respectively. Instantiation of code fragments in the algorithm skeleton is denoted by <...>.

```
schema(max F wrt X, C) :-
exists(first-derivative(F)),
symbolic_solve(d(F, X) == 0, Solution),
if (Solution != not_found)
C = "<X> := <Solution>";
else {schema(getStartValue(F,X), CInit);
schema(getStepsize(F,X),CSteplength);
schema(getStepdir(F,X),CStepdir);
C = "{<CInit>;
while(converging(<X>))
<X> := <X> +
<CSteplength>*<CStepdir>; }";
}.
```

Figure 10: Synthesis Schema (Fragment)

Schemas can also be extended in such a way that the annotations required for the certification are generated automatically.

Thus, the synthesized program can be assembled from various (and different) parts and algorithms. The output of the synthesis kernel is a program in a procedural intermediate language. The AUTOBAYES backend takes this intermediate code, optimizes it and generates code for the chosen target system.

Currently, we support Octave, Matlab and standalone C, but only small parts of the code generator are system-specific; therefore, new target systems can thus be added easily.

Certification procedures for high-quality data analysis code often mandate manual code inspections. These require that the code is readable and well documented. Human understandability is strong requirement, even for programs not subject to these procedures, as manual code manipulation is often necessary, e.g., for performance tuning or system integration. However, existing program generators often produce code that is hard to read and understand. In order to overcome this problem, AUTOBAYES generates thoroughly documented code: approximately one third of the output lines are automatically generated comments. These comments contain explanations of the crucial "synthesis decisions", e.g. which algorithm schema has been used. Also, model assumptions and proof obligations that could not be discharged during the synthesis are laid out clearly. In future versions of AUTOBAYES, we will extend this to produce detailed, standardized design documents along with the generated code. Furthermore, AUTOBAYES can generate code which generates artificial data for the mode, e.g. for visualization, simulation and testing purposes.[7]

The following figure shows the AutoBayes system architecture, extended for code certification.

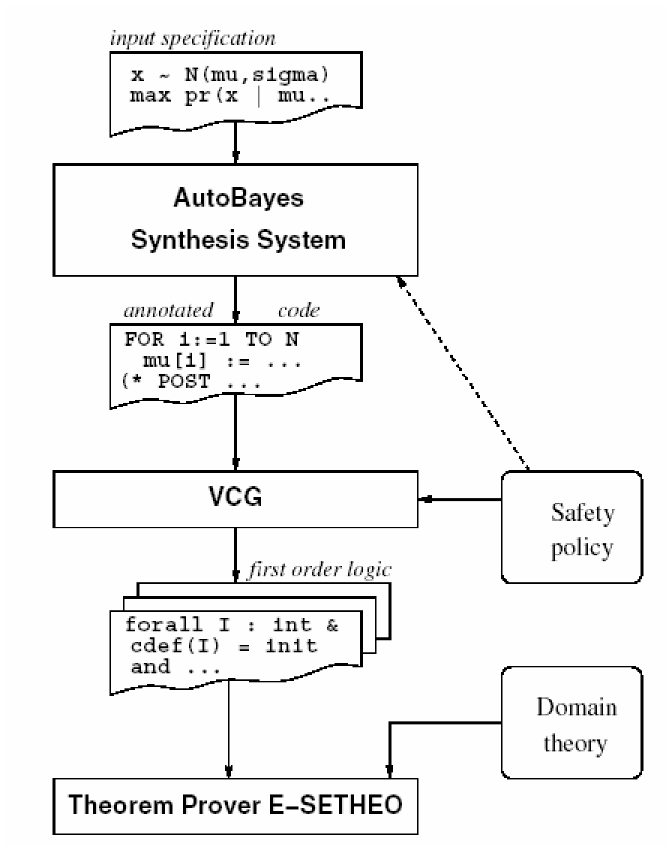


Figure 11: AutoBayes system architecture, extended for code certification[8]

6.1. For more information

For more information about software synthesis see the following resources:

- http://ase.arc.nasa.gov/schumann

- Johann M. Schumann. *Automated Theorem Proving in Software Engineering,* Springer Verlag, 2001, xiv+228 pages, ISBN 3-540-67989-8
- Johann Schumann, Bernd Fischer, Mike Whalen, and Jon Whittle.
  Certification Support for Automatically Generated Programs
  In Proc. HICSS'36, 2003
- Johann Schumann, Bernd Fischer, Mike Whalen, and Jon Whittle.
  Certification Support for Automatically Generated Programs
  In Proc. HICSS'36, 2003
- Bernd Fischer and Johann Schumann
  AutoBayes: A System for Generating Data Analysis Programs from Statistical Models.
  Journal Functional Programming, 2002 (in print)
- Mike Whalen, Johann Schumann, and Bernd Fischer.
  Synthesizing Certified Code.
  In Proceedings FME 2002, LNAI, Springer, 2002.
- Mike Whalen, Johann Schumann, and Bernd Fischer.
  Combining Program Synthesis with Automatic Code Certification (System Description)
  Conference on Automated Deduction (CADE) }, LNAI, Springer, 2002.
- Bernd Fischer and Johann Schumann
  Automated Synthesis of Statistical Data Analysis Programs
  Proc. Workshop SDP (Science Data Processing) 2002, NASA Goddard, 2002.

6.2. How Synthesis Tools Compare To Traditional V&V Tools

6.2.1. Coverage

Coverage relates to how much of the code can be tested. Generally, test plans cannot achieve 100% coverage because the complexity of code makes it difficult to think of all necessary test cases. Even if all test cases could be conceived, it might take many years to complete them. Therefore, important code components and pathways through the code are tested.

Since properties of synthesized code are proven correct by mathematical means; coverage comes much closer to 100%.

6.2.2. Other testing strategies

The following table compares traditional testing techniques to those needed for software synthesis:

Traditional Testing Techniques	Synthesis
Individual test cases and test scripts	Certification of safety properties to ensure that some aspects of code are correct. Corresponds to 100% coverage but limited checking that output is correct. Also, synthesis makes it possible to automate generation of test cases from formal specification.
Code review	Preparation for code review can be simplified using sophisticated generated documentation Proof of software properties provides higher degree of confidence in synthesized software
Advanced testing including Static analysis	Code checked by safety policies
Simulation of scenarios	In the future, it may be possible to generate environment properties and simulation scenarios
Regression testing, Animation/Visualization	Regression testing containing automated test cases, as well as other test cases

6.3. Certification of Synthesis Tools

There are two common approaches to certifying V&V tools:

- Traditional certification process (such as the tool qualification required by DO-178B) where the V&V tool is rigorously tested until deemed correct by a review board. After certification the tool cannot be modified or enhanced without going through the same rigorous process and being re-certified. This is a time-consuming and expensive process. It can result in V&V tools which are quickly outdated as hardware improves or the complete lack of V&V tools for new types of software like the neural adaptive flight control system in the Intelligent Flight Control System (IFCS).
- Qualification of the tool "kernel" – This approach divides the tool into two parts: the kernel and the supporting code. For example, SKATE is an air traffic control tool containing a kernel, TSAFE (a collision avoidance system for aircraft), and supporting code for graphical user interface, etc. TSAFE was certified via traditional means and cannot be changed without re-certification. The supporting code can be enhanced for improved display equipment, etc. This approach results in trusted code for safety-critical functions and flexible code that can be improved to meet future needs in a cost effective fashion.

Synthesis tools fall easily into the latter category. The proof checker is the kernel to be certified via traditional means and the supporting code should remain flexible to cost effectively meet future needs as they arise.

7. ARTIFACTS

Artifacts are documents, code, presentations or other materials resulting from the software development process. Examples of artifacts include the Software Requirements Standards (SRS), Software Design Document (SDD), source and object code, et al. This section contains a list of common artifacts.

Examples of precise content for each artifact depend upon the governing standards for the project (i.e., in the United States, defense projects are subject to MIL-STD 498, aerospace projects are subject to DO-178B and/or NASA standards, etc.)

Artifacts should be updated regularly so they contain the most recent information. In fact, a successful presentation to the FAA for certification must include updated artifacts rather than original documentation with modification notifications.

Typical artifacts relevant to the product-oriented approach are listed below. New artifacts specific to software synthesis are highlighted in bold and proposed enhancements, if any, are described. Depending upon the project, not all these artifacts may be required.

- Requirements:
    - Software Requirements Standards (SRS) – traditional SRS is sufficient
    - Software Requirements Data - traditional software requirements data is sufficient
    - **Formal Requirements** – a new artifact containing a mathematical description of the software. Called a specification in AUTOBAYES.
- Plans:
    - Certification Plan – should include the process and tools for generating synthesized code and how the synthesized code will be integrated into the system. Should also describe the certification or proof including any applicable research. NASA/CR contains information about the Plan for Software Aspects of Certification (PSAC) required by DO-178B.
    - Software Development Plan (SDP) – should contain a description of software synthesis to be used for selected software components
    - Software Verification Plan (SVP) – should contain new techniques to verify software synthesis techniques
    - Software Test Plan – should include tests for synthesized code and integration of synthesized code into the system
    - Software Configuration Management Plan (SCMP) – should contain the version of synthesized code to be used
    - Software Quality Assurance Plan (SQAP)
- Design Documents:
    - Software Design Standards (SDS) – should contain standards for software synthesis
    - Software Code Standards – should contain code standards for software synthesis
    - Software Design Document - automatically generated by software synthesis tools for synthesized software components. A sample synthesized SDD is shown in Appendix C. Appendix D contains a comparison of the synthesized SDD to standard software design documents for MIL STD 498, IEEE/EIA 12207 and DO-178B. It also describes recommendations for enhancing standard design documents with the synthesized SDD.
- Configuration Management:

  - Software Configuration Index – should contain version of synthesized code
  - Software Configuration Management Records - should contain reports for the synthesis process
- Code:
  - Source Code – synthesized code will be generated automatically from formal requirements
  - Executable Object Code – should include synthesized code
- Results of V&V:
  - Problem Reports – not expected for synthesized code because synthesized software has been proven correct. Problem reports for synthesized code should require review of formal requirements.
  - Software Verification Cases and Procedures
  - Software Verification Results – can rely on the synthesis process to tie generated code to requirements. Should reduce manual tracking efforts.
  - Software Quality Assurance Records
- Summary:
  - Software Accomplishment Summary
- **Certificate** – a safety explanation of the mathematical proof from the proof checker described in Appendix E

8. APPENDIX A: ACRONYMS

Term	**Definition**
ANSI	American National Standards Institute
ARC	Ames Research Center
CM	Configuration Management
DFRC	Dryden Flight Research Center
FAA	Federal Aviation Administration
EIA	Electronic Industries Association
IEC	International Electro-technical Commission
IEEE	Institute of Electrical and Electronic Engineers
ISO	International Organization for Standardization
IV&V	(NASA) Independent Verification & Validation
JAA	Joint Aviation Authorities
JPL	Jet Propulsion Lab
MIL STD	Military Standard
NASA	National Aeronautics and Space Administration
NPD	NASA Policy Directive
NPG	NASA Procedures and Guidelines
RTCA	Requirements and Technical Concepts for Aviation
USA	United Space Alliance
V&V	Verification & Validation

Note: *More Acronyms: http://www.ksc.nasa.gov/facts/acronyms.html*

9. APPENDIX B: GLOSSARY

Artifact: Document, code, presentation or other materials resulting from the software development process. Examples of artifacts include the Software Requirements Standards (SRS), Software Design Document (SDD), source and object code, et al. Section 7 contains a list of common artifacts.

Black Box testing: Requirements-driven testing where engineers select system input and observe system output/reactions

Certifiable Software Synthesis: Technique for generating software from formal logic and establishing that it is correct based on mathematical proofs

Certification: Legal recognition by the certification authority that a software product complies with the requirements[9]

CSCI: Computer Software Configuration Item (a term used in NASA or Military standards to describe a product like a jet engine or a computer system)

Fidelity: Integrity of testbed. For example: low fidelity testbed may have a simulator rather than actual spacecraft hardware. The highest fidelity testbed is the actual hardware being tested

Flight Operational Readiness Review (ORR): Review by an independent team of engineers who have not worked on the project. Sometimes called Flight Readiness Review (FRR).

Mission critical: Loss of capability leading to possible reduction in mission effectiveness[10]

Modified Condition And Decision Coverage (MCDC): Defined as checking that *"every point of entry and exit in the program has been invoked at least once, every condition is a decision in the program has taken all possible outcomes at least once, every decision has been shown to independently affect that decision's outcome. A condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions."*[9]

Nominal: Expected behavior for no failure, for example: nominal behavior for a valve may be "open" or "shut."

Off-Nominal: Unexpected failure behavior, for example: off-nominal behavior for a valve may be "stuck open" or "stuck shut."

Process-oriented approach: The traditional process software must undergo in order to be approved or certified. This process is described in various standards, but is generally the same in that regulatory authorities will be looking for evidence that all potential hazards have been identified and that appropriate steps have been taken to deal with them.

Product-oriented approach: Viewing software components like widgets that can be proven correct via mathematical proofs rather than looking for evidence that all potential hazards have been identified and mitigated. Underwriter's Laboratory uses a product-oriented approach for certifying that fireproof doors will not burn.

Safety-critical: Failure or design error could cause a risk to human life[10]

Software Synthesis: Technique for generating software from formal logic

Test Readiness Review (TRR): Review by the internal project team

Validation: Process of determining that the requirements are correct and complete

Verification: Evaluation of results of a process to ensure correctness and consistency with respect to the input and standards provided to that process

White Box Testing: Design-driven testing where engineers examine internal workings of code

10. APPENDIX C: SAMPLE SYNTHESIZED SDD

The was generated automatically from the http://ase.arc.nasa.gov/autobayes/autobayes.html by following these instructions:

> Go to select specification and get the "mix-gaussians"
> Then you should see the specification in the edit window. Then press "submit" and after a while (and some messages) the system should say "done" and you can click at the "see design document"

10.1. Software Design Document

Module name:	mog
Module title:	Mixture of Gaussians
Date generated:	Thu Feb 20 17:09:57 2003
User:	143.232.64.118
Version of AutoBayes:	0.0.1

10.1.1. Summary

This document describes the specification, design and generation of code for the module mog. The code and this document has been automatically generated by the tool AutoBayes/AutoFilter.

This document has been generated automatically and should not be modified manually.

10.1.2. Input Specification

The following sections list and describe the input specification for the module mog. This input specification comprises the entire information which is provided by the user for the generation of the module mog. Other options, which can influence the operation of AutoBayes are entered via command-line options and are listed in the specification below.

The following section lists the textual input specification for the module mog. For details on the syntax and semantics of the input language see AutoBayes-input-language The subsequent section shows the Bayesian network which is underlying this input specification.

10.1.2.1. Textual Input Specification

```
/**AutoBayesFile************************************************************

  Filename         [ $Source: /home/schumann/CVS/PN/examples/mixture/mix-gaussians.ab,v $ ]

  Synopsis         [ Mixture of Gaussians ]

  Author     [
                     Wray Buntine
                     Bernd Fischer
                     Tom Pressburger
                     Johann Schumann
                    ]

Revision    [ $Id: mix-gaussians.ab,v 2.3 2001/11/21 18:35:18 fisch Exp $ ]

Description     [

      This is the "classical" finite mixture of Gaussians model. It
      assumes n_points observed data points x(.) which are generated by
      n_classes different normal distributions (i.e., classes); the
      generating class c(i) for a datapoint x(i) is hidden. The relative
      class frequencies are given by the unknown probability vector rho(.).
      The task is to summarize the classes by their mean values mu(.) and
      standard distributions sigma(.) as well as to estimate their
      frequencies rho(.).
      This model fixes the n_classes as 3 and specifies a tolerance
      which is appropriate for 3 classes and ~600 data points.

                    ]

References  [

   B. S. Everitt and D. J. Hand, Finite Mixture Distributions,
   Chapman & Hall, 1981.

            ]

See also    [ ]

Known Bugs  [

      The distribution statement for c should also allow the shorthands

```

```
          c(_) ~ discrete(rho).

        and

          c(_) ~ discrete(rho(0..n_classes-1)).

            ]

   Modification [

        $Log: mix-gaussians.ab,v $
        Revision 2.3  2001/11/21 18:35:18  fisch
        Added constraint on n_points.

        Revision 2.2  2001/08/22 00:32:20  fisch
        Minor syntactical changes.

        Revision 2.1  2000/11/07 21:12:17  fisch
        Completed documentation.

        Revision 2.0  2000/10/31 19:23:52  fisch
        CVS version cleanup

        Revision 1.1  2000/08/29 00:48:23  fisch
        initial revision in new syntax, renamed from mixture-gaussians.pl

        Revision 1.1  1999/10/22 18:51:13  schumann
        initial revision

                     ]


*************************************************************************/

model mog as 'Mixture of Gaussians'.

```

```
%
% Model parameters
%

const nat n_points as 'Number of data points'.
        where 0 < n_points.

const nat n_classes := 3 as 'Number of classes'.
        where 0 < n_classes.
        where n_classes << n_points.


%
% Class probabilities
%

double rho(0..n_classes-1).
        where 0 = sum(I := 0 .. n_classes-1, rho(I))-1.


%
% Class parameters
%

double mu(0..n_classes-1).

double sigma(0..n_classes-1).
        where 0 < sigma(_).


%
% Hidden variable
%

double c(0..n_points-1) as 'class assignment vector'.

```

```
%%% change into new notation:
%%% c(_) ~ discrete(rho) or c(_) ~ discrete(rho(0..n_classes-1))
c(_) ~ discrete(vector(I := 0 .. n_classes-1, rho(I))).


%
% Data
%

const double tolerance := 0.0003 as 'tolerance for appr. 600 data points'.

data double x(0..n_points-1).

x(I) ~ gauss(mu(c(I)), sigma(c(I))).


%
% Goal
%

max pr(x|{rho,mu,sigma}) for {rho,mu,sigma}.
```

11. APPENDIX D: COMPARISON OF SYNTHESIZED SDD TO STANDARDS

The following table contains a comparison of the Synthesized SDD shown in Appendix C to the software design guidelines found in IEEE 12207, MIL STD 498 and DO-178B.

The table is organized as follows:

- Synthesis Section – name and description of content for each section in the Synthesized SDD.
- IEEE 12207 Section – applicable IEEE 12207 section from 12207.1 Paragraph 6.16
- MIL STD 498 Section – applicable MIL STD 498 paragraph from the SOFTWARE DESIGN DESCRIPTION (SDD) Data Information Description (DID) Identification Number: DI-IPSC-81435
- DO-178B Section – applicable DO-178B paragraph

Table 2: Comparison of Synthesized SDD to Standards

Synthesis Section	**IEEE Section**	**MIL STD 498 Section**	**DO-178B Section**
Document Title	Paragraph 6.16.3a	Title page or identifier	Paragraph 11.10a
Module Name	Paragraph 6.16.3a and h (to prevent duplicate names, add an identification number along with the name)	Section 1.1 Scope Identification	Paragraph 11.10a
Module Title	Paragraph 6.16.3a	Section 1.1 Scope Identification	Paragraph 11.10a
Date Generated	Paragraph 6.16.3a	Section 1.1 Scope Identification	Paragraph 11.10i
User	Paragraph 6.16.3a	Section 1.1 Scope Identification	
Version of AutoBayes		Section 2 Referenced Documents	
Summary		Section 1.3 Document overview	
Input Specification	Paragraph 6.16.3b, c	Section 4 Architecture Design	Paragraph 11.10c
Code Generation Process	Paragraph 6.16.3f	Section 4.1 CSCI Components and Section 6 Requirements Traceability	Paragraph 11.10b, c, d, e, f, g, h, I and j
Intermediate Code			
Final Code	Paragraph 6.16.3e	Section 4.2 Concept of execution	Paragraph 11.10b, c, d, e, f, g, h, I and j
Compiler Warnings/Errors		*Note: Review would be required should any warnings or errors result from compilation to evaluate the risk associated with the error or warning*	

Unlike human generated code, synthesized code provides proof obligations to ensure that the generated software is accurate. This feature is above and beyond the content requirements of any current standards. Conclusions and recommendations specific to each standard are described in the following sections.

11.1. MIL STD 498

11.1.1. Conclusions

For military projects following MIL STD 498, the synthesized document may be best suited for use as an attachment for SDD Section 4.1, CSCI Components. The SDD would contain a brief description of the synthesized module with a reference to the synthesized document.

11.1.2. Recommendations

Should the intent be to generate a document that rigorously complies with MID STD 498 SDD content requirements, the Synthesis tool should be enhanced to include the following items:

- Table of Contents
- Page numbers
- System overview
- DID Section 2 - Referenced Documents: a list of upstream or downstream documents (in the Life Cycle) like Software Requirements or Specification documents. The reference to the “Version of AutoBayes” is also relevant to this section.
- Section 3 – CSCI-wide design decisions: Computer Software Configuration Item (CSCI) design decisions include how software will behave from the user’s point of view in meeting its requirements and other decisions affecting the selection and design of the software units (element in the design like an object, module, class or database).

  Design decisions that respond to requirements designated critical, such as those for safety, security, or privacy, shall be placed in separate subparagraphs.

  If a design decision depends upon system states or modes, this dependency shall be indicated. Design conventions needed to understand the design shall be presented or referenced.

  Examples of CSCI-wide design decisions are the following:

  1. Design decisions regarding inputs the CSCI will accept and outputs it will produce including interfaces with other systems and users. May reference the Interface Design Descriptions (IDDs).
  2. Design decisions on CSCI behavior in response to each input or condition, including actions the CSCI will perform, response times and other performance characteristics, description of physical systems modeled, selected equations/algorithms/rules, and handling of disallowed inputs or conditions.
  3. Design decisions on how databases/data files will appear to the user. May reference Database Design Descriptions (DBDDs)
  4. Selected approach to meeting safety, security, and privacy requirements
  5. Other CSCI-wide design decisions made in response to requirements, such as selected approach to providing required flexibility, availability, and maintainability.

- DID Section 4.1e: Add description of planned hardware resources (processor capacity, memory capacity, input/output device, communications/network equipment…) and utilization of hardware like typical usages, worst-case usage, assumption of certain events and any special considerations affecting utilization
- 4.3 Interface design – Add description of interface characteristics

11.2. IEEE 12207

11.2.1. Conclusions

For projects following IEEE 12207, the synthesized document may be best suited for use as an attachment for the SDD document unless fields could be added for data entry (or cut and paste) of missing information listed in the following section.

11.2.2. Recommendations

Should the intent be to generate a document that rigorously complies with IEEE 12207.1 Paragraph 6.16 content recommendations, the synthesis tool should be enhanced to include the following items:

- Static relationships of software units
- Rationale for software item design
- Reuse element identification (add a number to the module name)
- Define types of errors that are not specified in the software requirements and the handling of those errors
- Add Life cycle data characteristics per Annex H of IEEE/EIA 12207.0

11.3. DO-178B

11.3.1. Conclusions

For projects adhering to DO-178B, the synthesized document may be best suited for use as an attachment for the SDD document unless fields could be added for data entry (or cut and paste) of missing information listed in the following section.

11.3.2. Recommendations

Should the intent be to generate a document that rigorously complies with DO-178B Paragraph 11.10 content recommendations, the Synthesis tool should be enhanced to include the following items:

- Rationale for design decisions that are traceable to safety-related system requirements
- Resource limitations, the strategy for managing each resource and its limitations, the margins, and the method for measuring those margins, for example, timing and memory
- Scheduling procedures and inter-processor/inter-task communication mechanisms, including time-rigid sequencing, preemptive scheduling, Ada rendezvous and interrupts
- Details for their implementation, for example, software data loading, user-modifiable software, or multiple-version dissimilar software
- Partitioning methods and means of preventing partition breaches

12. APPENDIX E: SAFETY DOCUMENT GENERATOR

This appendix was written by Ewen Denney and Ram Prasad Venkatesan.

Formal Certification is the idea that a mathematical proof of some property of a piece of software can be regarded as a certificate of correctness which, in principle, can be subjected to external scrutiny. In practice, proofs themselves are unlikely to be of much interest to engineers. Moreover, formal mathematical proofs are unlikely to blend well with traditional certification approaches. However, it is possible to use the information obtained from a detailed mathematical analysis of some software to produce a simple textual report.

In the Automated Software Engineering group at NASA Ames, Ewen Denney and Ram Prasad Venkatesan have developed a Safety Document Generator (SDG) that automatically generates "safety reports" for each part of a synthesized program with respect to a given safety policy. The document generator is intended to be generic and currently supports two safety policies: safety with respect to array bounds and safety with respect to initialization of variables.

The document generator generates the explanations from the verification conditions generated by a so-called Verification Condition Generator (VCG). The verification conditions are used to identify every construct of the program that needs to be analyzed for safety, and provide an explanation for its safety with respect to a given safety property. A typical safety explanation traces the components of each relevant term to its ground definition within the program, explaining the safety of all the intermediate terms along the path. The user is also provided with the flexibility of restricting the explanations to the safety of specific lines or specific expressions in the program.

The safety document generator is intended to create safety documents for the code synthesized by the AutoBayes and AutoFilter systems which are being developed by the ASE group.

EXAMPLE:

Consider the following simple program:

```
0     proc(eg){

1     a[10] : int
2     b : int ;
3     c : int ;
4     d : int ;

5     b = 1 ;
6     c = 2 ;
7     d = b*b + c*c ;

8     for(i=0;i<10;i++)
      {
9       if(i < 5)
        {
10        a[d+i] = d ;
        }
          else
        {
11         a[2*d-1-i] = d ;
          }
        }
      }
```

12.1. SAFETY EXPLANATIONS

The following is a safety explanation for the above code:

```
ARRAY BOUNDS
------------

---------------------------------------------------------------------------
PROGRAM         : eg
SAFETY POLICY   : Array bounds
DATE            : 08-06-2003
---------------------------------------------------------------------------

 ARRAY ACCESSES IN THE PROGRAM

       a[d+i]           10
       a[2*d-1-i] 11


 SAFETY EXPLANATIONS

The access a[d+i] at line 10 (if the condition at line 9 is true) is
safe as the term d is evaluated from d=b*b+c*c at line 7; the term b
is evaluated from b=1 at line 5; the term c is evaluated from c=2 at
line 6; for each value of the loop index i from 0 to 9 at line 8; d+i
is within 0 and 9; and hence the access is within the bounds of the
array defined at line 1.


The access a[2*d-1-i] at line 11 (if the condition at line 9 is false)
is safe as the term d is evaluated from d=b*b+c*c at line 7; the term
b is evaluated from b=1 at line 5; the term c is evaluated from c=2 at
line 6; for each value of the loop index i from 0 to 9 at line 8;
2*d-1-i is within 0 and 9; and hence the access is within the bounds
of the array defined at line 1.

INITIALIZATION OF VARIABLES
----------------------------

---------------------------------------------------------------------------
PROGRAM         : eg
SAFETY POLICY   : Initialization of variables
DATE            : 08-06-2003
---------------------------------------------------------------------------

CONSTRUCTS CONSIDERED FOR INITIALIZATION SAFETY

b           5
c           6
d           7
i           8
i<5         9
```

```
a           10
a           11


 SAFETY EXPLANATIONS


The assignment b=1 at line 5 is safe;

The assignment c=2 at line 6 is safe;

The assignment d=b*b+c*c at line 7 is safe; the term b is initialized
from b=1 at line 5; the term c is initialized from c=2 at line 6;

The loop index i ranges from 0 to 9 and is initialized at line 8;

The conditional expression i<5 appears at line 9; the loop index i
ranges from 0 to 9 and is initialized at line 8;


The assignment a[d+i]=d at line 10 is safe (if the condition at line 9
is true) ; the term d is initialized from d=b*b+c*c at line 7; the
term b is initialized from b=1 at line 5; the term c is initialized
from c=2 at line 6; the loop index i ranges from 0 to 9 and is
initialized at line 8;


The assignment a[2*d-1-i]=d at line 11 is safe (if the condition at
line 9 is false) ; the term d is initialized from d=b*b+c*c at line 7;
the term b is initialized from b=1 at line 5; the term c is
initialized from c=2 at line 6; the loop index i ranges from 0 to 9
and is initialized at line 8;
```

13. REFERENCES

[1] Interview with Dale Mackall, Sr. Dryden Flight Research Center Verification and Validation engineer on January 16, 2003

[2] *Software Considerations in Airborne Systems and Equipment Certification,* Document No RTCA (Requirements and Technical Concepts for Aviation) /DO-178B, December 1, 1992. (Copies of this document may be obtained from RTCA, Inc., 1140 Connecticut Avenue, Northwest, Suite 1020, Washington, DC 20036-4001 USA. Phone: (202) 833-9339)

[3] Email from Dr. Ewen Denney dated April 4, 2003

[4] C. Kreitz. Program synthesis. In W. Bibel and P. H. Schmitt (eds.), *Automated Deduction - A Basis for Applications*, pp.105–134. Kluwer, 1998.

[5] B. Fischer and J. Schumann. AutoBayes: A system for generating data analysis programs from statistical models. *J. Functional Programming*, 2002. To appear. http://ase.arc.nasa.gov/people/fischer/.

[6] Bernd Fischer and Johann Schumann, *Automating the Analysis of Planetary Nebulae Images* In proceedings ASE 2003

[7] Bernd Fischer and Johann Schumann, *Automated Synthesis of Statistical Data Analysis Programs,* ASTEC seminar June 5, 2002

[8] Johann Schumann, Bernd Fischer, Mike Whalen, Jon Whittle, *Certification Support for Automatically Generated Programs,* Proceedings of the 36th Hawaii International Conference on System Sciences

[9] *Software Considerations in Airborne Systems and Equipment Certification,* Document No RTCA (Requirements and Technical Concepts for Aviation) /DO-178B, December 1, 1992. (Copies of this document may be obtained from RTCA, Inc., 1140 Connecticut Avenue, Northwest, Suite 1020, Washington, DC 20036-4001 USA. Phone: (202) 833-9339)

[10] Interview with Dale Mackall, Sr. Dryden Flight Research Center Verification and Validation engineer on January 16, 2003

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE February 2004	3. REPORT TYPE AND DATES COVERED Contractor Report

4. TITLE AND SUBTITLE Product-oriented Software Certification Process for Software Synthesis	5. FUNDING NUMBERS QSS NAS2-00065, 3006-NEL-001
6. AUTHOR(S) Stacy Nelson (Nelson Consulting)	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Nelson Consulting MS 269-1 Ames Research Center Moffett Field, CA 94035-1000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Washington, DC 20546-0001	10. SPONSORING/MONITORING AGENCY REPORT NUMBER NASA CR-2004-212819

11. SUPPLEMENTARY NOTES
Point of Contact: Stacy Nelson Ames Research Center, MS 269-1, Moffett Field, CA 94035-1000
(650) 604-3588

12a. DISTRIBUTION/AVAILABILITY STATEMENT Unclassified — Unlimited Subject Category 61 Distribution: Standard Availability: NASA CASI (301) 621-0390	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The purpose of this document is to propose a product-oriented software certification process to facilitate use of software synthesis and formal methods. Why is such a process needed? Currently, software is tested until deemed bug-free rather than proving that certain software properties exist. This approach has worked well in most cases, but unfortunately, deaths still occur due to software failure. Using formal methods (techniques from logic and discrete mathematics like set theory, automata theory and formal logic as opposed to continuous mathematics like calculus) and software synthesis, it is possible to reduce this risk by proving certain software properties. Additionally, software synthesis makes it possible to automate some phases of the traditional software development life cycle resulting in a more streamlined and accurate development process.

14. SUBJECT TERMS Software Certification, Aerospace Software, Verification and Validation;Software Safety, Software Development	15. NUMBER OF PAGES 48
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT