



The European Organisation for Civil Aviation Equipment
L'Organisation Européenne pour l'Équipement de l'Aviation Civile

FORMAL METHODS SUPPLEMENT TO ED-12C AND ED-109A

This document is the exclusive intellectual and commercial property of EUROCAE.

It is presently commercialised by EUROCAE.

This electronic copy is delivered to your company/organisation for internal use exclusively.

In no case it may be re-sold, or hired, lent or exchanged outside your company.

ED-216

January 2012

FORMAL METHODS SUPPLEMENT TO ED-12C AND ED-109A

This document is the exclusive intellectual and commercial property of EUROCAE.

It is presently commercialised by EUROCAE.

This electronic copy is delivered to your company/organisation for internal use exclusively.

In no case it may be re-sold, or hired, lent or exchanged outside your company.

ED-216

January 2012

FOREWORD

1. This document was prepared jointly by EUROCAE Working Group 71 (WG-71) and RTCA Special Committee 205 (SC-205), was approved by the Council of EUROCAE on 9 January 2012.
2. EUROCAE is an international non-profit making organisation in Europe. Membership is open to manufacturers and users of equipment for aeronautics, trade associations, national civil aviation administrations, and, under certain conditions, non-European organisations. Its work programme is principally directed to the preparation of performance specifications and guidance documents for civil aviation equipment, for adoption and use at European and world-wide levels.
3. The findings of EUROCAE are resolved after discussion amongst Members of EUROCAE and in collaboration with RTCA Inc, Washington D.C, through their appropriate committees.
4. EUROCAE performance specifications and other documents are recommendations only. EUROCAE is not an official body of the European Governments. Its recommendations are valid as statements of official policy only when adopted by a particular government or conference of governments.
5. Copies of this document may be obtained from:

EUROCAE
102, rue Etienne Dolet
92240 MALAKOFF
France

Telephone: 33 1 40 92 79 30

Fax: 33 1 46 55 62 65

Email: eurocae@eurocae.net

Website: www.eurocae.net

TABLE OF CONTENTS

FM.1.0	INTRODUCTION	1
	FM.1.1 PURPOSE	1
	FM.1.2 SCOPE	1
	FM.1.3 RELATIONSHIP TO OTHER DOCUMENTS	2
	FM.1.4 HOW TO USE THIS DOCUMENT	2
	FM.1.5 DOCUMENT OVERVIEW	2
	FM.1.6 CHARACTERISTICS OF FORMAL METHODS	2
	FM.1.6.1 Formal Models	3
	FM.1.6.2 Formal Analysis	4
	FM.1.7 FORMAL METHODS AS A VERIFICATION TECHNIQUE	5
FM.2.0	SYSTEM ASPECTS RELATING TO SOFTWARE DEVELOPMENT	6
FM.3.0	SOFTWARE LIFE CYCLE	7
FM.4.0	SOFTWARE PLANNING PROCESS	8
	FM.4.3 SOFTWARE PLANS	8
FM.5.0	SOFTWARE DEVELOPMENT PROCESSES	10
FM.6.0	SOFTWARE VERIFICATION PROCESS	12
	FM.6.1 PURPOSE OF SOFTWARE VERIFICATION	12
	FM.6.2 OVERVIEW OF SOFTWARE VERIFICATION PROCESS ACTIVITIES	14
	FM.6.2.1 Considerations for Formal Methods	14
	FM.6.3 SOFTWARE REVIEWS AND ANALYSES	14
	FM.6.3.1 Reviews and Analyses of High-Level Requirements	15
	FM.6.3.2 Reviews and Analyses of Low-Level Requirements	16
	FM.6.3.3 Reviews and Analyses of Software Architecture	17
	FM.6.3.4 Reviews and Analyses of Source Code	17
	FM.6.3.5 Reviews and Analyses of the Outputs of the Integration Process	18
	FM.6.3.6 Reviews and Analyses of the Formal Analysis Cases, Procedures, and Results for Requirements, Architecture, and Source Code	18
	FM.6.4 SOFTWARE TESTING	19
	FM.6.5 SOFTWARE VERIFICATION PROCESS TRACEABILITY	19
	FM.6.6 VERIFICATION OF PARAMETER DATA ITEMS	19
	FM.6.7 FORMAL ANALYSIS OF THE EXECUTABLE OBJECT CODE	19
	FM.6.7.1 Principles of Coverage Analysis When Using Formal Methods	24
	FM.6.7.2 Reviews and Analyses of the Formal Analysis Cases, Procedures, and Results for the Executable Object Code	25
FM.7.0	SOFTWARE CONFIGURATION MANAGEMENT PROCESS	26
FM.8.0	SOFTWARE QUALITY ASSURANCE PROCESS	27
FM.9.0	CERTIFICATION LIAISON PROCESS	28
FM.10.0	OVERVIEW OF CERTIFICATION PROCESS	29
FM.11.0	SOFTWARE LIFE CYCLE DATA	30
	FM.11.1 PLAN FOR SOFTWARE ASPECTS OF CERTIFICATION	30
	FM.11.3 SOFTWARE VERIFICATION PLAN	30
	FM.11.6 SOFTWARE REQUIREMENTS STANDARDS	30
	FM.11.7 SOFTWARE DESIGN STANDARDS	30
	FM.11.8 SOFTWARE CODE STANDARDS	31
	FM.11.13 SOFTWARE VERIFICATION CASES AND PROCEDURES	31
FM.12.0	ADDITIONAL CONSIDERATIONS	32

ANNEX FM.A	PROCESS OBJECTIVES AND OUTPUTS BY SOFTWARE LEVEL IN ED-12C	33
ANNEX FM.B	ACRONYMS AND GLOSSARY OF TERMS	44
ANNEX FM.C	PROCESS OBJECTIVES AND OUTPUTS BY ASSURANCE LEVEL IN ED-109A	46
APPENDIX FM.A – COMMITTEE MEMBERSHIP		57
APPENDIX FM.B – FREQUENTLY ASKED QUESTIONS AND DISCUSSION PAPERS		67

LIST OF FIGURES

FIGURE FM.6-1	LEVEL A SOFTWARE VERIFICATION PROCESSES	13
FIGURE FM.6-2	POSSIBLE VERIFICATION PATHS FOR EXECUTABLE OBJECT CODE WITH RESPECT TO SOURCE CODE AND REQUIREMENTS	23
FIGURE FM.B.1-1	SOFTWARE LIFE CYCLE FOR UNIT PROOF	73
FIGURE FM.B.1-2	PROOF PROCESS	77
FIGURE FM.B.1-3	OVERVIEW OF REQUIREMENT ARTIFACTS	82
FIGURE FM.B.1-4	SIMPLIFIED DISPLAY SYSTEM ARCHITECTURE	83
FIGURE FM.B.1-5	WINDOW MANAGER TOP LEVEL SOFTWARE MODEL	87
FIGURE FM.B.1-6	ANALYSIS PROCESS	88

LIST OF TABLES

TABLE FM.A-1	SOFTWARE PLANNING PROCESS	34
TABLE FM.A-2	SOFTWARE DEVELOPMENT PROCESSES	35
TABLE FM.A-3	VERIFICATION OF OUTPUTS OF SOFTWARE REQUIREMENTS PROCESS	36
TABLE FM.A-4	VERIFICATION OF OUTPUTS OF SOFTWARE DESIGN PROCESS	37
TABLE FM.A-5	VERIFICATION OF OUTPUTS OF SOFTWARE CODING & INTEGRATION PROCESSES	38
TABLE FM.A-6	TESTING OF OUTPUTS OF INTEGRATION PROCESS	39
TABLE FM.A-7	VERIFICATION OF VERIFICATION PROCESS RESULTS	40
TABLE FM.A-8	SOFTWARE CONFIGURATION MANAGEMENT PROCESS	41
TABLE FM.A-9	SOFTWARE QUALITY ASSURANCE PROCESS	42
TABLE FM.A-10	CERTIFICATION LIAISON PROCESS	43
TABLE FM.C-1	SOFTWARE PLANNING PROCESS	47
TABLE FM.C-2	SOFTWARE DEVELOPMENT PROCESSES	48
TABLE FM.C-3	VERIFICATION OF OUTPUTS OF SOFTWARE REQUIREMENTS PROCESS	49
TABLE FM.C-4	VERIFICATION OF OUTPUTS OF SOFTWARE DESIGN PROCESS	50
TABLE FM.C-5	VERIFICATION OF OUTPUTS OF SOFTWARE CODING & INTEGRATION PROCESSES	51
TABLE FM.C-6	TESTING OF OUTPUTS OF INTEGRATION PROCESS	52
TABLE FM.C-7	VERIFICATION OF VERIFICATION PROCESS RESULTS	53
TABLE FM.C-8	SOFTWARE CONFIGURATION MANAGEMENT PROCESS	54
TABLE FM.C-9	SOFTWARE QUALITY ASSURANCE PROCESS	55
TABLE FM.C-10	SOFTWARE APPROVAL PROCESS	56
TABLE FM.B.1-1	REDUNDANCY MANAGEMENT SOFTWARE VERIFICATION RESULTS ..	71
TABLE FM.B.1-2	WINDOW MANAGER REQUIREMENTS FORMALIZED IN CTL	86
TABLE FM.B.1-3	WINDOW MANAGER ANALYSIS DATA	89

FM.1.0

INTRODUCTION

Formal methods are mathematically based techniques for the specification, development, and verification of software aspects of digital systems. The mathematical basis of formal methods consists of formal logic, discrete mathematics, and computer-readable languages. The use of formal methods is motivated by the expectation that, as in other engineering disciplines, performing appropriate mathematical analyses can contribute to establishing the correctness and robustness of a design. For example, formal methods, because of their mathematical basis, are capable of:

- Unambiguously describing requirements of software systems.
- Enabling precise communication between engineers.
- Providing verification evidence such as consistency and accuracy of a formally specified representation of software.
- Providing verification evidence of the compliance of one formally specified representation with another.

Formal methods are also capable of demonstrating properties of software systems such as:

- Freedom from exceptions.
- Freedom from deadlock.
- Non-interference between different levels of criticality.
- Worst case execution time.
- Bounds on stack size during execution.
- Freedom from unintended function.
- Correct synchronous or asynchronous behavior.

Since the publication of ED-12B, advances and experience have been gained in formal methods, their application, and tools. This supplement provides guidance for applicants and certification or approval authorities to facilitate the use of formal methods.

The adoption of formal methods into an established set of processes for development and verification can be an evolutionary refinement rather than an abrupt change of methodology. The extent to which formal methods are used to satisfy the objectives of ED-12C can vary according to aspects such as preferences of the program management, choice of techniques, and availability of specialized resources. Formal methods might be used in a very selective manner to partially address a small set of objectives, or might be the primary source of evidence for the satisfaction of many of the objectives concerned with development and verification.

FM.1.1

PURPOSE

This supplement identifies the modifications and additions to ED-12C objectives, activities, explanatory text, and software life cycle data that should be addressed when formal methods are used as part of the software life cycle. This includes the artifacts that would be expressed using some formal notation and the verification evidence that could be derived from them.

FM.1.2

SCOPE

This supplement discusses the use of formal methods in the software life cycle for software that is produced in accordance with ED-12C. If the applicant is not planning to use formal analysis as defined in FM.1.6, then the applicant does not need to comply with this supplement.

FM.1.3 RELATIONSHIP TO OTHER DOCUMENTS

This document supplements the guidance given in ED-12C and ED-109A. This document may be used in conjunction with any ED-12C-related supplements or other documents.

FM.1.4 HOW TO USE THIS DOCUMENT

This supplement should be used with and in the same way as ED-12C; however, the following should be noted:

- Section FM.1 contains explanatory text to aid the reader in understanding formal methods, and therefore is not to be taken as guidance.
- Annex FM.A of this supplement describes how the ED-12C objectives are revised in line with this formal methods guidance.
- Guidance from ED-12C should be used for all aspects of the software life cycle, apart from where formal methods are used. All guidance specific to formal methods is contained within this supplement.
- This supplement can be applied to ED-109A in the same manner as ED-12C. Applicants complying with ED-109A should be aware that the text within this supplement is based on ED-12C; thus, the ED-109A applicant needs to ensure that the proper ED-109A guidance is addressed. Annex FM.B of this supplement describes how the ED-109A, Annex A objectives are revised in line with the formal methods guidance in this supplement.
- In addition to the guidance contained in this supplement, a Discussion Paper is provided to clarify the guidance (see Appendix FM.B). The Discussion Paper contains no new or additional guidance.
- Major sections are numbered as FM.X.0 throughout this supplement. It should be noted that references to an entire section are identified as “section FM.X”; whereas, references to the content between section headers FM.X.0 and FM.X.1 are referred to as “section FM.X.0”.

FM.1.5 DOCUMENT OVERVIEW

This supplement is intended to be used with ED-12C. Sections FM.4, FM.5, FM.6, FM.11, FM.12, and Annex FM.A provide additional and modified text to the referenced sections of ED-12C. This supplement minimizes any redundancy between ED-12C and the document itself. Each affected section of ED-12C is included in this supplement. Where the guidance from ED-12C applies unchanged, this is stated in this supplement. Text taken directly from ED-12C is shown in italics.

In Annex FM.A, only those tables with changes are included. These tables include new objectives and changes to referenced text for existing objectives.

FM.1.6 CHARACTERISTICS OF FORMAL METHODS

Understanding some of the fundamental characteristics of formal methods and their scope of applicability is essential to appropriate application of the guidance in this supplement. Application of formal methods usually involves two types of activities: modeling and analysis, where the models and analysis procedures are derived from or defined by an underlying mathematically precise foundation. Although modeling and analysis are common activities in many software engineering efforts, these terms have particular meanings when used in the context of formal methods described in this supplement. Not all models or all analyses will meet requirements for being formal. Throughout this supplement, the terms formal models and formal analysis are used to distinguish those activities when they meet requirements specified herein for formality. A formal model combined with a formal analysis constitutes a formal method. If the applicant plans to use formal models but not formal analyses, then they do not need to comply with this supplement.

FM.1.6.1 Formal Models

Establishing a formal model of the software artifact of interest is fundamental to all formal methods. In general a model is an abstract representation of a given set of aspects of the software that is used for analysis, simulation, and/or code generation. In the context of this supplement, to be formal, a model should have an unambiguous, mathematically defined syntax and semantics. This makes it possible to use automated means to obtain guarantees that the model has certain specified properties.

Formal models can be produced by the development process (for example, requirement or design activities) or by formal analysis of the software artifact. Creating a formal model by formal analysis of a software artifact is possible when the artifact itself exhibits the characteristics of well-defined syntax and semantics, such as Source Code and Executable Object Code.

Throughout the development life cycle, different formal models may be used for different kinds of analysis and for the establishment of different properties. Examples of formal models include:

- Graphical models where the components of the diagram and the connections between them have mathematically defined syntax and semantics. For example, including but not limited to control diagrams or state machines.
- Textual based models that have a mathematically defined syntax and semantics. Example could be: Z, logic, set theory, algebraic or differential equations, subsets of C, subsets of Ada or other languages.
- Abstract models formed inside a formal methods tool by an analysis of the software artifacts of interest such as the code.

The applicability of formal methods to any particular software development activity is bounded by the ability to construct an appropriate formal model. A single model, or even a combination of models, may not be sufficient to capture all of the aspects of the software that are required to be verified according to the objectives of ED-12C. That is, in many cases, it is not possible to completely model the software using a formal notation. Requirements specified in natural language (such as English) may include properties that cannot be verified with a formal method. Models can also be insufficiently detailed to allow meaningful analysis of some properties and yet be perfectly adequate for others.

Despite these limitations, formal models can be a means of completely and accurately describing important software properties. Formal analyses can then be applied to provide assurance of these properties, and combined with other verification techniques, (for example, testing) to achieve satisfaction of all of the verification objectives. Formal methods provide comprehensive assurance of particular properties only for those aspects that are formalized in the formal model, so defining the limits of the model is essential.

Formal methods may be applied in many stages of the development life cycle, and to satisfy various verification objectives. Wherever formal models are used, they should be verified to provide assurance that they are an accurate representation of the artifact. This means that for the properties being analyzed, if they hold for the model then they hold for the actual artifact.

NOTE: *One method for ensuring the **correct** development of models is the use of approved ED-12C guidance for model-based development.*

FM.1.6.2 Formal Analysis

Although there are important benefits in creating formal models of life cycle artifacts, the most powerful benefits of formal methods are in the formal analysis of those models. Formal analysis can provide guarantees or proofs of software properties and compliance with requirements. Proof, or guarantee, implies that all execution cases are taken into account, achieving exhaustive verification. To conduct a formal analysis, a set of software properties is necessary. These properties are either created or embedded in specific tools that implement the formal analysis (for example, worst-case execution timing). Properties for formal analysis can be seen as analogous in many ways to verifiable requirements for testing. In testing, test cases should be easily derivable from verifiable requirements; for formal analysis, the properties are usually also the verification cases. It may be possible to define a complete set of properties for a set of requirements. In this case, proof that the set of properties always holds true demonstrates the equivalence of the item under analysis to its requirements.

For the purpose of this supplement, an analysis method can only be regarded as formal analysis if its determination of a property is sound. Sound analysis means that the method never asserts a property to be true when it is not true. The converse case, the assertion that a property is false when it may be true, colloquially “the raising of false alarms”, is a usability issue but not a soundness issue. Furthermore, it is acceptable for a method to return “don’t know” or not to return an answer when trying to establish whether a property holds, in which case additional verification is necessary.

There are many different kinds of formal analysis, but they can typically be classified in three categories: (1) deductive methods, such as theorem proving, (2) model checking, and (3) abstract interpretation.

- (1) Deductive methods involve mathematical arguments, such as mathematical proofs, for establishing a specified property of a formal model. A correct proof of a property provides rigorous evidence of that property for the formal model. These proofs are typically constructed using an automated or interactive theorem proving system. Even with such assistance, constructing proofs can be difficult, or impossible in some cases. However, once a proof is completed, automatic checking of the correctness of that proof is usually trivial.
- (2) Model checking explores all possible behaviors of a formal model to determine whether a specified property is satisfied. In cases where the property is not satisfied, a counter-example illustrating where and how the property fails to hold is generated automatically. In some cases, a model checker may not be able to determine whether the given property is satisfied or not.
- (3) Abstract interpretation is a theory for formally constructing conservative representations (that is, enforcing soundness) of the semantics of programming languages. In practice, this method is used for constructing semantics-based analysis algorithms for the automatic, static, and sound determination of dynamic properties of infinite-state programs. With abstract interpretation, a formal model is generated, usually within a tool, which is specific to the particular property under analysis. It can be viewed as a partial execution of a computer program which determines specific effects of the program (for example, control structure, flow of information, stack size, number of clock cycles) without actually performing all the calculations.

All of these formal analyses share two important attributes. First, the methods rely on formal models. Compliance between artifacts can never be demonstrated between an informal model and a formal model, using formal analysis. For example, using formal methods to demonstrate compliance between specification and code requires that both be formal models. Second, all of the formal analyses are generally implemented using a tool. Any tool that supports the formal analysis should be assessed under the tool qualification guidance required by ED-12C and qualified where necessary.

Scalability of formal analysis can be an important issue because formal analysis examines all possible behavior of the formal model. As the size of the model grows, the computing power required to analyze the model may become exponentially greater.

FM.1.7**FORMAL METHODS AS A VERIFICATION TECHNIQUE**

Formal methods may enable complete exploration of possible behaviors of a formal model with respect to certain properties. Subject to the formal model employed being an accurate representation of the artifact (which should be established by additional verification, as per guidance in this supplement), formal methods provide assurance that certain kinds of faults are not present at the level of description and stage of the life cycle considered by the formal model. This supplement provides guidance for using this assurance to meet verification objectives of ED-12C. Some objectives may be fully met using formal methods. Other objectives may require additional verification to supplement formal methods, such as complementary testing in the integrated target computer environment.

FM.2.0

SYSTEM ASPECTS RELATING TO SOFTWARE DEVELOPMENT

Section 2 of ED-12C is unchanged.

FM.3.0

SOFTWARE LIFE CYCLE

Section 3 of ED-12C is unchanged.

FM.4.0

SOFTWARE PLANNING PROCESS

ED-12C sections 4.1, 4.2, and 4.4 to 4.6 are unchanged. Some life cycle data items, referred to in section 4.5 of ED-12C, have been modified to address formal notations.

The applicant should ensure that the intended use of this supplement is acceptable to the appropriate certification authority. Several supplements may be applied to an applicant's software life cycle. The applicant should plan an approach that will comply with the objectives of ED-12C and those applicable objectives from the supplements. As such, all relevant supplements and the supplements' associated objectives should be considered. If the applicant's particular approach identifies a conflict between the objectives, then the applicant should propose a solution to the conflict in the Plan for Software Aspects of Certification (PSAC). The applicant may use a single set of software plans or multiple set of software plans addressing the different supplement. Using a single set of software plans, containing sections addressing each supplement, will provide an integrated approach with a clear and comprehensive plan.

As supplements only extend guidance from ED-12C for a specific technology, the applicant should first consider all of the ED-12C objectives, and then the supplement's objectives, and lastly any other additional considerations. One approach is to consolidate all objectives of ED-12C and the applicable supplements, and for each objective provide a statement of how compliance will be achieved, along with identifying any applicable life cycle data items. Because the objective numbering scheme is unique between the Annex A Tables of ED-12C and the Annex A Tables of the supplements, the identification of the objectives and their document sources should be clear and unambiguous.

FM.4.3

SOFTWARE PLANS

The software plans define the means of satisfying the objectives of ED-12C and other applicable documents. They specify the organizations that will perform those activities. The software plans are:

- *The Plan for Software Aspects of Certification (ED-12C section 11.1) serves as the primary means for communicating the proposed development methods to the certification authority for agreement, and defines the means of compliance. The Plan for Software Aspects of Certification gives an overview of how and where formal methods will be used and what evidence those methods will provide.*
- *The Software Development Plan (ED-12C section 11.2) defines the software life cycle(s), and software development environment, and the means by which the software development process objectives will be satisfied. The Software Development Plan details the precise development steps that will include the use of formal methods.*
- *The Software Verification Plan (ED-12C section 11.3) defines the means by which the software verification process objectives will be satisfied. The Software Verification Plan details the methods used to gather the verification evidence including that obtained by formal analysis. All assumptions related to the formal analysis should be described and justified, especially those associated with the target computer. The effects of these assumptions on the ability to detect errors and verify functionality using formal analysis should be considered, and additional software verification processes and methods should be specified in the Software Verification Plan. Consideration should be given as to how the various verification techniques, such as testing and formal analysis, are combined to meet all the verification objectives for the software. Specifically the overlap or potential gaps between the methods need to be resolved, and may affect the integration test planning. This is necessary as formal methods can be applied to partially meet objectives or fully meet objectives for a subset of the software. They can also be used to verify all of the software for specific objectives.*

- There is no additional guidance for formal methods related to the Software Configuration Management and Software Quality Assurance Plans (ED-12C sections 11.4 and 11.5).

The remainder of ED-12C section 4.3 is unchanged.

FM.5.0

SOFTWARE DEVELOPMENT PROCESSES

This section discusses the objectives and activities of the software development processes. The software development processes are applied as defined by the software planning process (ED-12C section 4) and the Software Development Plan (ED-12C section 11.2). ED-12C Table A-2 of Annex A is a summary of the objectives and outputs of the software development processes by software level. The software development processes are:

- *Software requirements process.*
- *Software design process.*
- *Software coding process.*
- *Integration process.*

Software development processes produce one or more levels of software requirements. High-level requirements are produced directly through analysis of system requirements and system architecture. Usually, these high-level requirements are further developed during the software design process, thus producing one or more successive, lower levels of requirements. However, if Source Code is generated directly from high-level requirements, then the high-level requirements are also considered low-level requirements and the guidance for low-level requirements also apply.

NOTE: *The applicant may be required to justify software development processes that produce a single level of requirements.*

The development of software architecture involves decisions made about the structure of the software. During the software design process, the software architecture is defined and low-level requirements are developed. Low-level requirements are software requirements from which Source Code can be directly implemented without further information.

Each software development process may produce derived requirements. Some examples of requirements that might be determined to be derived requirements are:

- *The need for interrupt handling software to be developed for the chosen target computer.*
- *The specification of a periodic monitor's iteration rate when not specified by the system requirements allocated to software.*
- *The addition of scaling limits when using fixed point arithmetic.*

High-level requirements may include derived requirements, and low-level requirements may include derived requirements. In order to determine the effects of derived requirements on the system safety assessment and system requirements, all derived requirements should be made available to the system processes including the system safety assessment process.

Using formal models to define requirements or design artifacts allows some of the verification objectives to be satisfied by the use of formal analysis.

Formalizing requirements and/or design may increase the effort required to specify them, but, generally, additional errors are found and removed earlier during this process due to the additional scrutiny and standardization.

Not all of the requirements need to be defined formally when using a formal method. The key to using a formal method is that there is sufficient definition in the formal model to verify properties about the specific artifacts being analyzed. For example, if a formal model is created at the architecture level, then the software architecture can be analyzed independently of low-level requirements. In the same way, if a formal model is created at the low-level requirements level, then low-level requirements can be analyzed independently of the high-level requirements and software architecture. Some methods create a formal model at the source code level, embedding information flow from the design into the code, which is then checked by formal analysis. It may even be possible to apply formal methods to analyze properties of object code. In this case, object code becomes a formal model, the semantics of which are treated the same by the formal analysis as they are by the target hardware.

ED-12C sections 5.1 to 5.5 are unchanged.

FM.6.0

SOFTWARE VERIFICATION PROCESS

This section discusses the objectives and activities of the software verification process. Verification is a technical assessment of the outputs of the software planning process, software development processes, and the software verification process. The software verification process is applied as defined by the software planning process (ED-12C section 4) and the Software Verification Plan (ED-12C section 11.3). See ED-12C section 4.6 for the verification of the outputs of the planning process.

Verification is not simply testing. Testing, in general, cannot show the absence of errors. As a result, the following sections use the term "verify" instead of "test" to discuss the software verification process activities, which are typically a combination of reviews, analyses, and tests. Guidance for reviews, analyses (other than formal), and testing is given in ED-12C.

With formal analyses, the correctness of life cycle data with respect to a formal model can be proved or disproved. The extent to which formal analysis is used to satisfy the verification objectives can vary according to the ability to construct appropriate formal models, the choice of analysis techniques, and the availability of tools and related resources. Formal analysis might be used to partially address a small set of the verification objectives, or it might be the primary source of evidence for the satisfaction of most of the objectives concerned with verification. As a minimum, testing will still be required to ensure that the Executable Object Code is compatible with the target computer.

FM.6.1 PURPOSE OF SOFTWARE VERIFICATION

The purpose of the software verification process as described in ED-12C section 6.1 is unchanged. To help explain where formal analysis fits in the software verification process as described in the following sections (FM.6.2–FM.6.6), it is helpful to look at the specific verification objectives. [Figure FM.6-1](#) shows the relationships for Level A among life cycle data items, development activities, and verification activities with respect to objectives in ED-12C Tables A-3 through A-6. Each verification activity is depicted by a dashed line and the objectives which it satisfies are annotated on the arrows. The life cycle data items are shown as hexagons. Solid lines are used to signify development activities.

NOTE: *Not all life cycle data items involved in verification are shown on the diagram. For example, standards are taken as implicit for activities such as "Conformance to standards." Verification of outputs of software testing as defined in ED-12C Table A-7 is also not shown.*

As shown in [Figure FM.6-1](#), some of the verification objectives appear repeatedly. For example, there are objectives for traceability and compliance between the high-level requirements and systems requirements, between the low-level requirements and high-level requirements, and between Source Code and low-level requirements. Formal analysis can be used to satisfy many of the verification objectives, completely in some cases and only partly in others. In this last case, the verification plan should describe how the combination of formal analysis and other methods satisfies the objective completely.

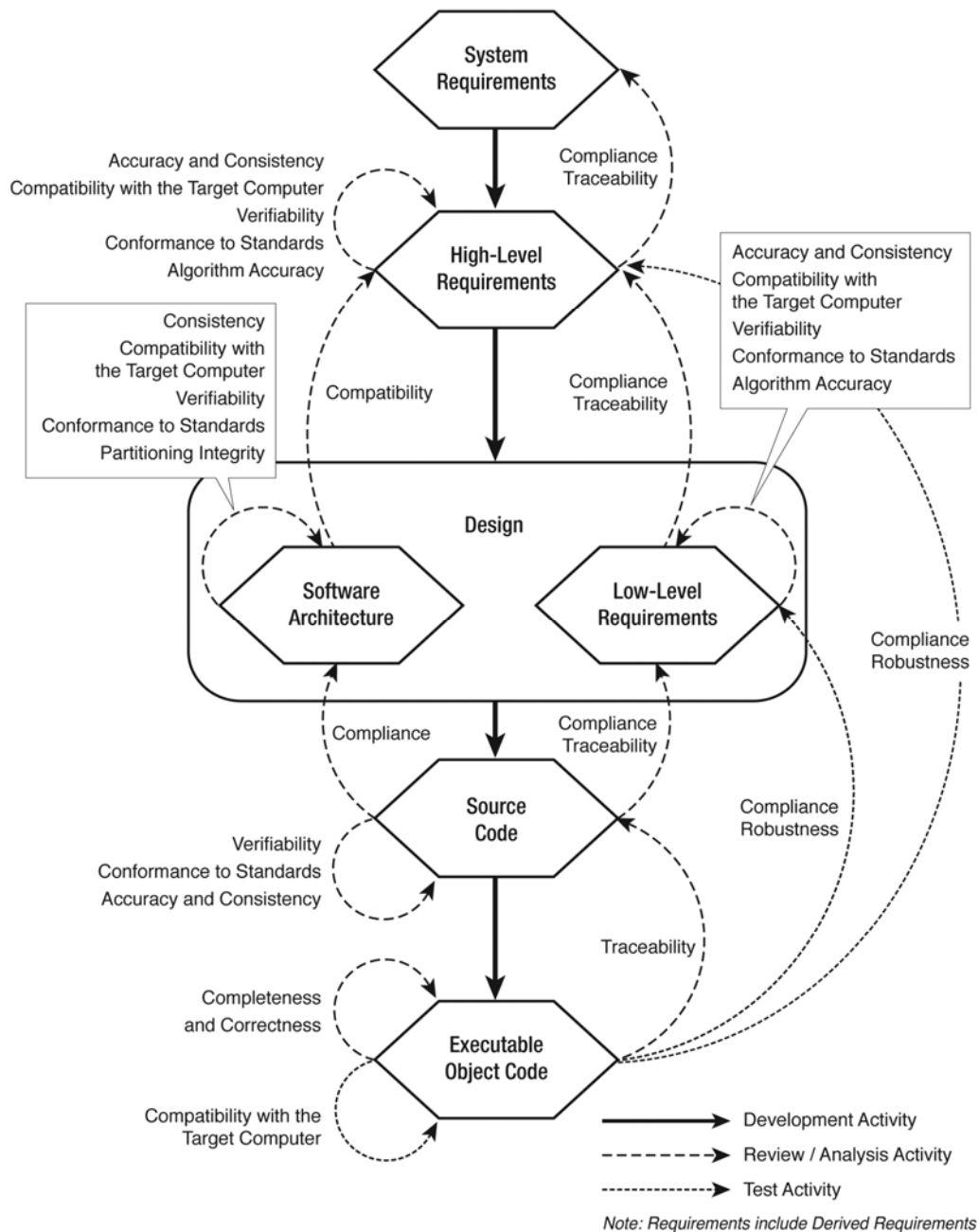


FIGURE FM.6-1: LEVEL A SOFTWARE VERIFICATION PROCESSES
(ED-12C TABLES A-3 TO A-6)

FM.6.2 OVERVIEW OF SOFTWARE VERIFICATION PROCESS ACTIVITIES

ED-12C section 6.2 is unchanged, except for the addition of section FM.6.2.1.

FM.6.2.1 Considerations for Formal Methods

When formal analysis is used to meet a verification objective, there is an additional objective to ensure that each formal method used is correctly defined, justified, and appropriate to meet this verification objective. Activities for this are as follows:

- a. All notations used for formal analysis should be verified to have precise, unambiguous, mathematically defined syntax and semantics; that is, they are formal notations.
- b. The soundness of each formal analysis method should be justified. A sound method never asserts that a property is true when it may not be true.
- c. All assumptions related to each formal analysis should be described and justified; for example, assumptions associated with the target computer or about the data range limits.

FM.6.3 SOFTWARE REVIEWS AND ANALYSES

Reviews and analyses are applied to the outputs of the software development processes. One distinction between reviews and analyses is that analyses provide repeatable evidence of correctness and reviews provide a qualitative assessment of correctness. A review may consist of an inspection of an output of a process guided by a checklist or similar aid. An analysis may examine in detail the functionality, performance, traceability, and safety implications of a software component, and its relationship to other components within the system or equipment.

Formal analysis can be used to satisfy the objectives for software reviews and analysis as follows:

- a. Compliance: If the input life cycle data items and the output life cycle data items of a software development process are formally modeled, then formal analysis can be used to verify compliance. However, formal methods cannot show that a derived requirement is correctly defined and has a reason for existing; this should be achieved by review.
- b. Accuracy: Formal notations are precise and unambiguous, and can be used to demonstrate accuracy of the representation of a life cycle data item.
- c. Consistency: Life cycle data items that are formally expressed can be checked for consistency (the absence of conflicts). If a set of formal statements is found to be logically inconsistent, then the inconsistencies should be addressed before any subsequent analysis is conducted.
- d. Compatibility with the target computer: Formal analysis can be used to detect potential conflicts between a formal description of the target computer and the life cycle data item.
- e. Verifiability: Being able to express a requirement in the formal notation defined in the software verification plan is evidence of verifiability because it is unambiguous and has a mathematically defined syntax and semantics and then could be formally analyzed.
NOTE: *Requirements involving “always/never” cannot generally be verified by a finite set of test cases, but may be verified by formal analysis.*
- f. Conformance to standards: Life cycle data items that are formally expressed should be compliant with any standards defining the formalism.

NOTE: *Invalid results will be obtained if ill-formed requirements are allowed. Since formal notations have a precise, unambiguous, and mathematically defined syntax, automated syntax checkers are appropriate for verifying that the formally stated requirements are well-formed with respect to syntax. In addition, an automated checker may enforce other restrictions on the notation (for example, complexity of notational constructs and other design constructs that would not comply with the system safety objectives). Automated checking will need to be supplemented by review for those standards not amenable to automated checking.*

- g. Traceability: Traceability ensures that all input requirements have been developed into lower-level requirements or Source Code. Traceability from the inputs of a process to its outputs can be demonstrated by verifying with a formal analysis that the outputs of the process fully satisfy its inputs. Traceability from the outputs of a process to its inputs can be demonstrated by verifying with a formal analysis that each output data item is necessary to satisfy some input data item.
- h. Algorithm aspects: If life cycle data items are formally modeled, then algorithmic aspects can be checked using formal analysis.
- i. Requirement formalization correctness: If a requirement has been translated to a formal notation as the basis for using a formal analysis, then review or analysis should be used to demonstrate that the formal statement is a conservative representation of the informal requirement.

NOTE: *If the gap between an informal statement of the requirement and its embodiment in a formal notation is too large, then this may be difficult to review. The preciseness of formal notations is only an advantage when they maintain fidelity to the intent of the informal requirement.*

FM.6.3.1 Reviews and Analyses of High-Level Requirements

These review and analysis activities detect and report requirements errors that may have been introduced during the software requirements process. These verification activities confirm that the high-level requirements satisfy the objectives that follow. A mixture of both formal and informal requirements can be used for either or both system and high-level requirements. Formal methods can be used to meet all or part of the following objectives:

- a. Compliance with system requirements: *The objective is to ensure that the system functions to be performed by the software are defined, that the functional, performance, and safety-related requirements of the system are satisfied by the high-level requirements, and that derived requirements and the reason for their existence are correctly defined.* If the system requirements and high-level requirements are formally modeled, then formal analysis can be used to show compliance. Formal methods cannot show that derived requirements and the reason for their existence are correctly defined; this should be achieved by review.
- b. Accuracy and consistency: *The objective is to ensure that each high-level requirement is accurate, unambiguous, and sufficiently detailed, and that the requirements do not conflict with each other.* If the high-level requirements are expressed in a formal notation, then they will be precise and unambiguous. The formal model of high-level requirements can be checked for consistency (the absence of conflicts) and may enable accuracy checks to be carried out using formal analysis.
- c. Compatibility with the target computer: *The objective is to ensure that no conflicts exist between the high-level requirements and the hardware/software features of the target computer, especially system response times and input/output hardware.* If the high-level requirements and hardware/software features of the target computer are formally modeled, then potential conflicts can be detected through formal analysis.

- d. Verifiability: The objective is to ensure that each high-level requirement can be verified. Being able to express a high-level requirement in the formal notation defined in the Software Verification Plan is evidence of verifiability.
- e. Conformance to standards: The objective is to ensure that the Software Requirements Standards were followed during the software requirements process and that deviations from the standards are justified. High-level requirements expressed in a formal notation can be checked for conformance to standards by using formal analysis.
- f. Traceability: The objective is to ensure that the functional, performance, and safety-related requirements of the system that are allocated to software were developed into the high-level requirements. If formal methods are used to meet the objective of section FM.6.3.1, item a, this can provide evidence in support of traceability.
- g. Algorithm aspects: The objective is to ensure the accuracy and behavior of the proposed algorithms, especially in the area of discontinuities. If high-level requirements are formally modeled, then they can be checked for accuracy and behavior by using formal analysis.

FM.6.3.2 Reviews and Analyses of Low-Level Requirements

These review and analysis activities detect and report requirements errors that may have been introduced during the software design process. Reviews and analyses may be used to confirm that the software low-level requirements satisfy the objectives below. A mixture of both formal and informal requirements can be used for either or both high-level and low-level requirements. Formal methods can be used to meet all or part of the following objectives:

- a. Compliance with high-level requirements: The objective is to ensure that the low-level requirements satisfy the high-level requirements and that derived requirements and the design basis for their existence are correctly defined. If the high-level requirements and low-level requirements are formally modeled, then formal analysis can be used to show compliance. Formal methods cannot show that derived requirements and the reason for their existence are correctly defined; this should be achieved by review.
- b. Accuracy and consistency: The objective is to ensure that each low-level requirement is accurate and unambiguous, and that the low-level requirements do not conflict with each other. If the low-level requirements are expressed in a formal notation, then they will be precise and unambiguous. The formal model of low-level requirements can be checked for consistency (the absence of conflicts) and may enable accuracy checks to be carried out using formal analysis.
- c. Compatibility with the target computer: The objective is to ensure that no conflicts exist between the low-level requirements and the hardware/software features of the target computer, especially the use of resources such as bus loading, system response times, and input/output hardware. If the low-level requirements and hardware/software features of the target computer are formally modeled, then potential conflicts can be detected through formal analysis.
- d. Verifiability: The objective is to ensure that each low-level requirement can be verified. Being able to express a low-level requirement in the formal notation defined in the Software Verification Plan is evidence of verifiability.
- e. Conformance to standards: The objective is to ensure that the Software Design Standards were followed during the software design process and that deviations from the standards are justified. Low-level requirements expressed in a formal notation can be checked for conformance to standards by using formal analysis.
- f. Traceability: The objective is to ensure that the high-level requirements and derived requirements were developed into the low-level requirements. If formal methods are used to meet the objective of section FM.6.3.2, item a, this can provide evidence in support of traceability.

- g. Algorithm aspects: The objective is to ensure the accuracy and behavior of the proposed algorithms, especially in the area of discontinuities. If low-level requirements are formally modeled, then they can be checked for accuracy and behavior by using formal analysis.

FM.6.3.3 Reviews and Analyses of Software Architecture

These review and analysis activities detect and report errors that may have been introduced during the development of the software architecture. Reviews and analyses may be used to confirm that the software architecture satisfies the objectives below. A mixture of both formal and informal requirements can be used for either or both high-level requirements and software architecture. Formal methods can be used to meet all or part of the following objectives:

- a. Compatibility with the high-level requirements: The objective is to ensure that the software architecture does not conflict with the high-level requirements, especially functions that ensure system integrity, for example, partitioning schemes. If the high-level requirements and software architecture are formally modeled, then formal analysis can be used to show compatibility.
- b. Consistency: The objective is to ensure that a correct relationship exists between the components of the software architecture. This relationship exists via data flow and control flow. If the interface is to a component of a lower software level, it should also be confirmed that the higher software level component has appropriate protection mechanisms in place to protect itself from potential erroneous inputs from the lower software level component. If the software architecture is expressed in a formal notation, then it will be precise and unambiguous. Data and control flow between components can be checked for correct relationships using formal analysis.
- c. Compatibility with the target computer: The objective is to ensure that no conflicts exist, especially initialization, asynchronous operation, synchronization, and interrupts, between the software architecture and the hardware/software features of the target computer. If the software architecture and hardware/software features of the target computer are formally modeled, then potential conflicts can be detected through formal analysis.
- d. Verifiability: The objective is to ensure that the software architecture can be verified, for example, there are no unbounded recursive algorithms. Being able to express the software architecture in the formal notation as defined in the Software Verification Plan is evidence of verifiability.
- e. Conformance to standards: The objective is to ensure that the Software Design Standards were followed during the software design process and that deviations to the standards are justified, for example, deviations to complexity restrictions and design construct rules. Software architecture expressed in a formal notation can be checked for conformance to standards by using formal analysis.
- f. Partitioning integrity: The objective is to ensure that partitioning breaches are prevented. If the software architecture is formally modeled, some aspects of partitioning integrity can be verified by formal analysis.

FM.6.3.4 Reviews and Analyses of Source Code

These review and analysis activities detect and report errors that may have been introduced during the software coding process. Reviews and analyses can be used to confirm that the outputs of the software coding process are accurate, complete and can be verified. Primary concerns include correctness of the code with respect to the software requirements and the software architecture, and conformance to the Software Code Standards. If formal source code semantics exist, compliance with some or all of the following objectives can be achieved through formal methods.

- a. Compliance with the low-level requirements: The objective is to ensure that the Source Code is accurate and complete with respect to the low-level requirements and that no Source Code implements an undocumented function. If the low-level requirements are formally modeled and formal source code semantics exist, then formal analysis can be used to show compliance.

- b. Compliance with the software architecture: The objective is to ensure that the Source Code matches the data flow and control flow defined in the software architecture. If the software architecture is formally modeled and formal source code semantics exist, then data and control flow can be analyzed using formal analysis.
- c. Verifiability: The objective is to ensure the Source Code does not contain statements and structures that cannot be verified and that the code does not have to be altered to verify it. When Source Code has mathematically defined syntax and semantics and is consistent with the formal analysis defined in the Software Verification Plan, then this objective is satisfied.
- d. Conformance to standards: The objective is to ensure that the Software Code Standards were followed during the development of the code, for example, complexity restrictions and code constraints. Complexity includes the degree of coupling between software components, the nesting levels for control structures, and the complexity of logical or numeric expressions. This analysis also ensures that deviations to the standards are justified. Source Code with mathematically defined syntax and semantics can be checked for conformance to standards by using formal analysis.
- e. Traceability: The objective is to ensure that the low-level requirements were developed into Source Code. When formal methods are used to meet the objective of section FM.6.3.4, item a, this can provide evidence in support of traceability.
- f. Accuracy and consistency: The objective is to determine the correctness and consistency of the Source Code, including stack usage, memory usage, fixed point arithmetic overflow and resolution, floating-point arithmetic, resource contention and limitations, worst-case execution timing, exception handling, use of uninitialized variables, cache management, unused variables, and data corruption due to task or interrupt conflicts. The compiler (including its options), the linker (including its options), and some hardware features may have an impact on the worst-case execution timing and this impact should be assessed. If mathematically defined syntax and semantics exist for the Source Code, then these characteristics can be checked using formal analysis. The mathematically defined syntax and semantics may need to take into account the programming language standards, compiler information (for example, default behavior and configuration options), and characteristics of the target computer.

FM.6.3.5 Reviews and Analyses of the Outputs of the Integration Process

ED-12C section 6.3.5 is unchanged.

FM.6.3.6 Reviews and Analyses of the Formal Analysis Cases, Procedures, and Results for Requirements, Architecture, and Source Code

When formal analysis is used to meet a verification objective, it is necessary to ensure that the formal analysis was developed and performed accurately and completely for that objective. The topics should include:

- a. Formal analysis cases: The objective is to provide evidence that the formal analysis covers the objectives in sections FM.6.3.1–FM.6.3.4. All assumptions that were included in the formal analysis should be justified and any false assumptions (which would invalidate the analysis) should be identified and addressed.
- b. Formal analysis procedures: The objective is to verify that the formal analysis cases were accurately developed into formal analysis procedures and expected results.
- c. Formal analysis results: The objective is to ensure that the formal analysis results are correct and that discrepancies between actual and expected results are explained.

FM.6.4 SOFTWARE TESTING

If formal methods are being used to verify Executable Object Code, then FM.6.7 should be used. ED-12C section 6.4 applies if formal methods are not being used to verify Executable Object Code.

NOTE: *The text of FM.6.7 is derived from ED-12C section 6.4, with modification for formal analysis. The text in FM.6.7 is not italicized because there is no corresponding section in ED-12C.*

FM.6.5 SOFTWARE VERIFICATION PROCESS TRACEABILITY

Software verification process traceability activities include:

- a. *Trace Data, showing the bi-directional association between software requirements and verification cases, is developed. This Trace Data supports the requirements-based coverage analysis.*
- b. *Trace Data, showing the bi-directional association between verification cases and verification procedures, is developed. This Trace Data enables verification that the complete set of verification cases has been developed into verification procedures.*
- c. *Trace Data, showing the bi-directional association between verification procedures and verification results, is developed. This Trace Data enables verification that the complete set of verification procedures has been executed.*

FM.6.6 VERIFICATION OF PARAMETER DATA ITEMS

ED-12C section 6.6 is unchanged.

FM.6.7 FORMAL ANALYSIS OF THE EXECUTABLE OBJECT CODE

Verification of the Executable Object Code is primarily performed by testing. This can be assisted by reviews and analyses as indicated in ED-12C. Formal analyses exist to establish specific properties such as worst-case execution time and stack usage. These analyses can replace some testing aspects of the Executable Object Code, but not all of them. Tests executed in target hardware are always required to ensure that the software in the target computer will satisfy the high-level requirements as defined in ED-12C section 6.4.3, item a.

Verification activities for the Executable Object Code are used to demonstrate that the software satisfies its requirements and to demonstrate with a high degree of confidence that errors which could lead to unacceptable failure conditions, as determined by the system safety assessment process, have been removed.

The verification activities should verify:

- Correct operation of the software in the target computer environment.
- Correct inter-relationships between software requirements and components and the correct implementation of the software requirements and software components within the software architecture. If a combination of formal methods and testing has been planned by the applicant, then verification of these interrelationships should be achieved by integration testing as stated in ED-12C.
- Correct implementation of software low-level requirements.

NOTE: *The above list is in order of precedence; that is, you may be able to achieve all three by verification in the target computer environment.*

When using formal methods:

- (1) Formal analysis cases should be based primarily on the software requirements.
- (2) Formal analysis cases should be developed to verify correct functionality and to establish conditions that reveal potential errors.
- (3) Software requirements coverage analysis should determine what software requirements were not verified.
- (4) Structural coverage analysis should determine what software structures were not verified.

Guidance for requirements-based formal analysis case selection includes:

- Normal range and robustness (abnormal range) cases should be considered.
- The specific cases should be developed from the software requirements and the error sources inherent in the software development processes.

The objectives for verification of the Executable Object Code include:

- a. Compliance with the high-level requirements: The objective is to demonstrate the ability of the software to respond to normal inputs and conditions. Activities for normal range cases include:

- Real and integer input variables should be considered using valid boundary values.
- For time-related functions, such as filters, integrators and delays, multiple iterations of the code should be performed to check the characteristics of the function in context.
- For state transitions, analyses should be developed to consider the transitions possible during normal operation.
- For software requirements expressed by logic equations, the normal range cases should verify the variable usage and the Boolean operators.

Formal methods can be used to satisfy this objective as described in FM.6.7.f.

- b. Robustness for high-level and low-level requirements: The objective is to demonstrate the ability of the software to respond to abnormal inputs and conditions. Activities for robustness cases include:

- Real and integer variables should be considered.
- System initialization should be considered during abnormal conditions.
- The possible failure modes of the incoming data should be determined, especially complex, digital data strings from an external system.
- For loops where the loop count is a computed value, formal analysis cases should be developed to attempt to compute out-of-range loop count values, and thus demonstrate the robustness of the loop-related code.
- A check should be made to ensure that protection mechanisms for exceeded frame times respond correctly.
- For time-related functions, such as filters, integrators and delays, formal analysis cases should be developed for arithmetic overflow protection mechanisms.
- For state transitions, formal analysis cases should be developed to provoke transitions that are not allowed by the software requirements.

Formal methods can be used to satisfy this objective as described in FM.6.7, item f.

- c. Verification of software integration: The verification activity should concentrate on the inter-relationships between the software requirements, and on the implementation of requirements by the software architecture. The objective is to ensure that the software components interact correctly with each other and satisfy the software requirements and software architecture. Typical errors to consider include:

- Incorrect initialization of variables and constants.
- Parameter passing errors.
- Data corruption, especially global data.
- Inadequate end-to-end numerical resolution.
- Incorrect sequencing of events and operations.

Formal methods can be used to verify the absence of some of the error types listed above, as described in FM.6.7, item f. Additionally, testing will be required to verify the software integration in target hardware.

- d. Compliance with the low-level requirements: The objective is to demonstrate the ability of the software to respond to normal inputs and conditions. Activities for normal range cases include:

- Real and integer input variables should be considered using valid boundary values.
- For time-related functions, such as filters, integrators and delays, multiple iterations of the code should be performed to check the characteristics of the function in context.
- For state transitions, analyses should be developed to consider the transitions possible during normal operation.
- For software requirements expressed by logic equations, the normal range cases should verify the variable usage and the Boolean operators.

Formal methods can be used to satisfy this objective as described in FM.6.7, item f.

Typical errors to consider include:

- Failure of an algorithm to satisfy a software requirement.
- Incorrect loop operations.
- Incorrect logic decisions.
- Failure to process correctly legitimate combinations of input conditions.
- Incorrect responses to missing or corrupted input data.
- Incorrect handling of exceptions, such as arithmetic faults or violations of array limits.
- Incorrect computation sequence.
- Inadequate algorithm precision, accuracy or performance.

- e. Compatibility with the target computer: This verification activity should concentrate on error sources associated with the software operating within the target computer environment, and on the high-level functionality. The objective is to ensure that the software in the target computer will satisfy the high-level requirements. This is achieved primarily by testing. Typical errors to consider include:

- Incorrect interrupt handling.
- Failure to satisfy execution time requirements.
- Incorrect software response to hardware transients or hardware failures, for example, start-up sequencing, transient input loads and input power transients.
- Data bus and other resource contention problems, for example, memory mapping.
- Inability of built-in test to detect failures.

- Errors in hardware/software interfaces.
- Incorrect behavior of control loops.
- Incorrect control of memory management hardware or other hardware devices under software control.
- Stack overflow.
- Incorrect operation of mechanism(s) used to confirm the correctness and compatibility of field-loadable software.
- Violations of software partitioning.

Formal methods can be used to verify the absence of some of the errors listed above providing that all of the following are satisfied:

- (1) A formal model of the Executable Object Code with respect to the error condition is defined. This model should be complete with respect to the necessary hardware characteristics for the specific error type (for example, as long as all types of usage of the stack and instances of usage of the stack are modeled and the available stack size is defined, then a stack overflow check is achievable).
- (2) Formal evidence demonstrates the absence of some of the errors listed above in the Executable Object Code, either by computing safe upper bounds of resource consumption (for example stack usage, worst-case execution timing) or by proving formally expressed properties (for example absence of resource contention).

f. Verification of Property Preservation Between Source and Executable Object Code

Options for verification activities of Executable Object Code include:

- (1) Formal analysis of Executable Object Code can be used to satisfy the objectives if all of the following conditions are satisfied:
 - The requirements are formally expressed.
 - A formal model of the Executable Object Code is defined.
 - Formal evidence demonstrates that the formal model of the Executable Object Code satisfies the requirements.
- (2) Formal analysis of Source Code can be used to satisfy the objectives if all of the following conditions are satisfied:
 - The requirements are formally expressed.
 - A formal model of the Source Code is defined.
 - Formal evidence demonstrates that the formal model of the Source Code satisfies the requirements.
 - Complementary analysis shows property preservation between the Source Code and the EOC. By verifying the correctness of the translation of source to object code, formal analysis performed at the Source Code level against high or low-level requirements can be used to infer correctness of the Executable Object Code against high or low-level requirements. This is similar to the way that coverage metrics gained from Source Code can be used to establish the adequacy of tests to verify the target system. If a compiler, linker, or other means generates additional code that is not directly traceable to Source Code statements, then additional verification should be performed to establish property preservation.

Figure FM.6-2 shows that where formal methods are used at source level to provide evidence of the compliance and robustness of the Executable Object Code to the high level or low-level requirements, the verification of property preservation between Source Code and Executable Object Code should be demonstrated. The diagram shows alternative verification paths; for example, formal methods could show compliance between Executable Object Code and low-level requirements (via property preservation in the object code), instead of testing, and testing could show compliance between Executable Object Code and high-level requirements.

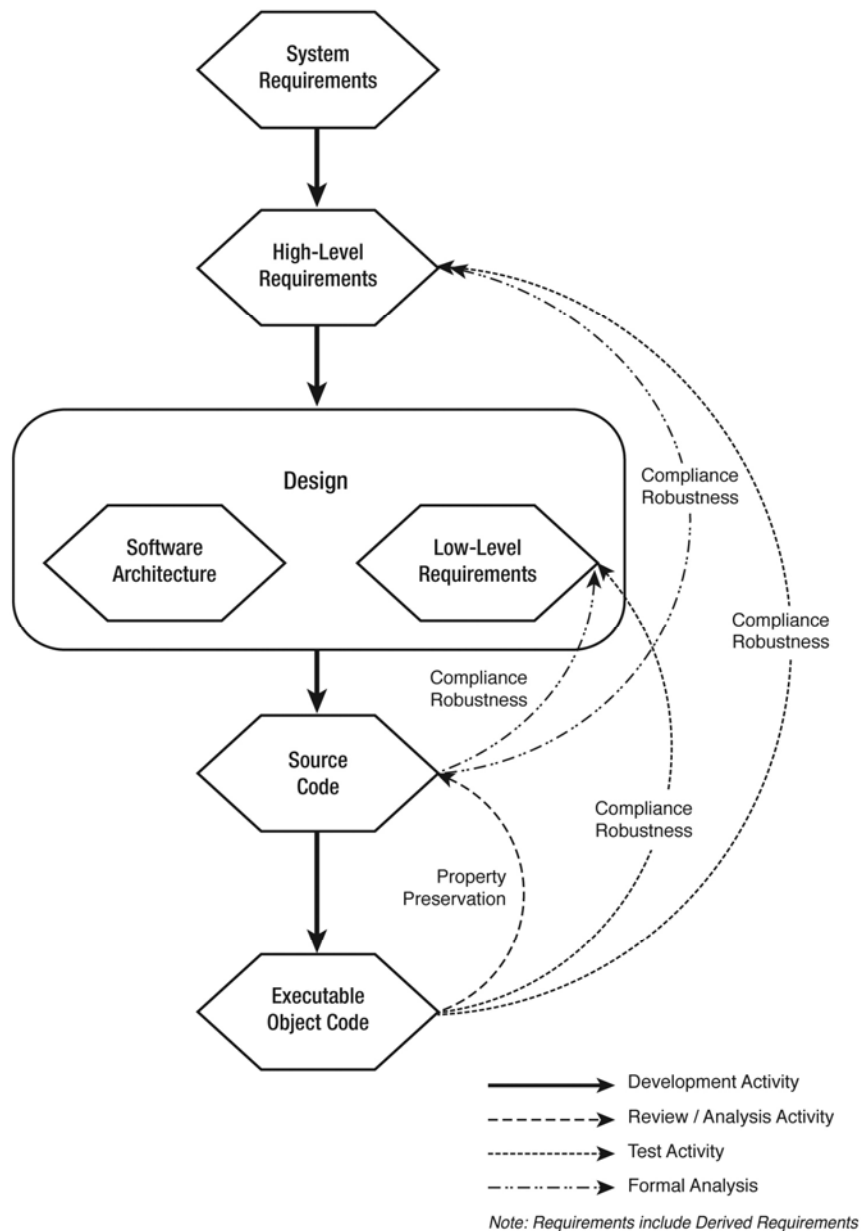


FIGURE FM.6-2: POSSIBLE VERIFICATION PATHS FOR EXECUTABLE OBJECT CODE WITH RESPECT TO SOURCE CODE AND REQUIREMENTS

FM.6.7.1 Principles of Coverage Analysis When Using Formal Methods

Test coverage analysis is a two step process, involving requirements-based coverage analysis and structural coverage analysis. Requirements-based coverage establishes that verification evidence exists for all of the requirements for the system. Structural coverage analysis, however, is driven out of the impracticality of achieving exhaustive testing and the consequent need to establish metrics to ensure that the testing performed is rigorous and sufficient. Unlike testing, the use of formal methods may provide an exhaustive assessment of the software. However, additional activities are required to achieve comparable coverage analysis.

When any low-level testing is used to verify low-level requirements for a software component, then the entire software component will be subject to ED-12C testing guidance for structural coverage analysis.

When only formal methods are used to verify requirements for a software component, sections FM.6.7.1.1-FM.6.7.1.5 describe activities that are required for coverage analysis of the component.

There are three objectives for coverage analysis. The first two objectives are for requirements-based coverage analysis which is very similar to the guidance in ED-12C and is detailed below in section FM.6.7.1.1:

- a. Coverage of high-level requirements is achieved.
- b. Coverage of low-level requirements is achieved.

The third objective is for structural coverage analysis:

- c. Verification coverage of software structure is achieved.

Structural coverage analysis is a notion strongly connected with testing and execution of code and is designed primarily to detect shortcomings in test. Verification by formal methods can be complete and exhaustive; hence equivalence to structural coverage is required rather than the analysis of how much structural coverage has been achieved. This equivalence is obtained by combining the activities described in sections FM.6.7.1.2 to FM.6.7.1.5 to detect:

- Shortcomings in requirements-based verification cases or procedures (covered in FM.6.7.1.2).
- Inadequacies in software requirements (covered in FM.6.7.1.3 and FM.6.7.1.4).
- Extraneous code, including dead code, and deactivated code (covered in FM.6.7.1.5).

FM.6.7.1.1 Requirements-Based Coverage Analysis

The purpose of requirements-based coverage analysis is to determine how well the implementation of the software requirements has been verified. This coverage analysis may reveal the need for additional requirements-based verification. Activities include:

- a. Analysis, using the associated Trace Data, to confirm that formal analysis cases exist for each software requirement.
- b. Analysis to confirm that formal analysis cases satisfy the criteria of normal range and robustness criteria as defined in sections FM.6.7, items a and b.
- c. Resolution of any deficiencies identified in the analysis. Possible solutions are adding or enhancing formal analysis cases.

FM.6.7.1.2 Complete Coverage of Each Requirement

The purpose is to ensure that all assumptions made during the formal analysis are verified. If the assumptions are all verified, then the formal analysis can give complete coverage of each requirement.

FM.6.7.1.3 Completeness of the Set of Requirements

Where requirements are formally modeled, it should be demonstrated that the set of requirements is complete with respect to the intended functions. That is:

- For all input conditions, the required output has been specified.
- For all outputs, the required input conditions have been specified.

If the set of requirements is found to be incomplete, then additional requirements (generally not derived requirements) should be added. If a demonstration of completeness cannot be achieved, then structural coverage analysis should be used.

FM.6.7.1.4 Detection of Unintended Dataflow Relationships

Verifying that the information flow in the Source Code complies with the requirements ensures that there are no unintended dependencies between the inputs and outputs of the code, and therefore ensures that each element in the Source Code is specified by requirements. If unintended dependencies exist then these should be resolved either by adding the missing requirements (generally not derived requirements) or removing the erroneous code.

FM.6.7.1.5 Detection of Extraneous Code, including Dead Code, and Deactivated Code

In addition to the formal method used, extraneous code, including dead code, and deactivated code should be identified by review or analysis (other than formal). Dead code should be removed and an analysis performed to assess the effect and the need for reverification. For deactivated code which is not intended to be executed in any configuration used within an aircraft or engine, a combination of analysis and testing should show that the means by which such code could be inadvertently executed are prevented, isolated, or eliminated. For deactivated code which is only executed in certain configurations of the target computer environment, the operational configuration needed for normal execution of this code should be established and additional verification cases and procedures developed to satisfy the required coverage objectives. Extraneous code should be removed or additional requirements specified to support the identified function or module.

FM.6.7.2 Reviews and Analyses of the Formal Analysis Cases, Procedures, and Results for the Executable Object Code

The review and analysis activities detect and report errors that may have been introduced within the formal analysis data. These reviews and analyses confirm that the formal analyses satisfy these objectives:

- a. Formal analysis cases: The objective is to provide evidence that the formal analysis achieves the coverage required for the applicable software level. All assumptions that were included in the formal analysis should be justified and any false assumptions (which would invalidate the analysis) should be identified and addressed. Formal analysis cases should be reviewed against the criteria given in section FM.6.7, items a and b.
- b. Formal analysis procedures: The objective is to verify that the formal analysis cases were correctly developed into formal analysis procedures and expected results.
- c. Formal analysis results: The objective is to ensure that the formal analysis results are correct and that discrepancies between actual and expected results are explained.