INTERNATIONAL STANDARD

ISO/IEC 23643

First edition
2020-06

Software and systems engineering — Capabilities of software safety and security verification tools

Ingénierie du logiciel et des systèmes — Capacités des outils de vérification de la sûreté et de la sécurité des logiciels

Reference number
ISO/IEC 23643:2020(E)

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents) or the IEC list of patent declarations received (see http://patents.iec.ch).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html.

This document was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 7, *Software and systems engineering*.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html.

Introduction

Since a few decades, the importance of software safety and security verification tools has increased for several reasons: 1) rapidly increasing complexity of software applications and systems, 2) increasing number of safety-critical systems through growing integration between software applications and systems (e.g. in critical infrastructures), 3) the rapid increase of the number of cyber threats, and 4) the urgent needs of safety in high and medium critical software-driven systems (e.g. transportation, energy production, Internet of Things (IoT), and general purpose Operating Systems and middleware). Additionally, the number of products and system development cases, where the origin of all software components to be used is not exactly known, even for open-source applications, is increasing and thus making safety and security verification and validation (V&V) essential.

This document restricts its point of view to software and excludes computing and any other hardware from the context. In these other domains, other V&V methods and tools are used.

It is important to realize that verification of safety and security of software does not necessarily verify the system safety and system security of a system using the software as a component. However, if a system consists of software components which are not verified, the safety and security of the system cannot be guaranteed at any level.

"Continuous everything", including continuous software development and thus versioning delivery, requires continuous software safety and security verification. At every new version, V&V needs to be redone. The popular "agile development processes" permit shorter development iterations and more frequent product delivery, and consequently this requires more frequent verification than traditional development approaches. Verification is needed during software development as well as during software maintenance, whenever safety or security of software can be endangered.

Validation answers the question "are we building the right product?"

Verification answers the question "are we building the product right?"

Software validation checks if the software product satisfies the intended use, such as defined in requirements and specifications. In other words (ISO/IEC 17029): "purpose of validation is to find out, whether a declared information (claim) is plausible". Software verification checks if the specifications and requirements are met either by running the software (testing) or by reviewing its artefacts (specification, model, or pseudo code). The latter can consist of animating or analyzing statically one of its artefacts. ISO/IEC 17029 defines that the "purpose of verification is to find out, whether a declared information (claim) is truthful". This document does not concern testing but animating and analyzing the artefacts, because "testing tools" is already well covered by and is the subject of ISO/IEC 30130.

This document is prepared as one of the series of single tool capabilities which are used with ISO/IEC 20741.

This document defines capabilities of and requirements for software safety and security verification tools.

This document is independent of the target application domains, as the languages, methods and associated tools are of general purpose, and can fit into different kinds of problems (e.g. functional specification languages can be used for any functional program).

Software and systems engineering — Capabilities of software safety and security verification tools

1 Scope

This document specifies requirements for the vendors and gives guidelines for both the users and the developers of software safety and security verification tools. The users of such tools include, but are not limited to, bodies performing verification and software developers who need to be aware and pay attention to safety and/or security of software. This document guides the verification tool vendors to provide as high-quality products as possible and helps the users to understand the capabilities and characteristics of verification tools.

This document introduces use cases for software safety and security verification tools and entity relationship model related to them. This document also introduces tool categories for software safety and security verification tools and gives category specific guidance and requirements for the tool vendors and developers.

2 Normative references

There are no normative references in this document.

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at http://www.electropedia.org/

3.1
application domain
well-defined set of applications

3.2
capability
quality of being able to perform a given activity

[SOURCE: ISO 19439:2006, 3.5]

3.3
certificate
attestation document issued by an independent third-party certification body

[SOURCE: ISO 22222:2005, 3.2]

3.4
defect
fault, or deviation from the intended level of performance of a system or *software* (3.19)

3.5
dynamic program analysis
process of evaluating a *software* (3.19) system or component based on its behaviour during execution

Note 1 to entry: The definition means that the software shall actually be compiled and run on a certain number of input data test cases. The physical response from the system is then examined and compared to expected results. Dynamic program analysis can be done manually or using an automated process. The results are examined either manually (e.g. with small input test data) or automatically using oracles.

3.6
entity
data concept that may have attributes and relationships to other entities

[SOURCE: ISO/TR 25100:2012, 2.1.3, modified — Note 1 to entry has been removed.]

3.7
evaluator
competent person engaged in the *verification* (3.33) or *validation* (3.32) of a system or *software* (3.19)

3.8
false negative
true *defect* (3.4) that has not been observed

Note 1 to entry: The term is used for analysis tools producing defect information during the analysis of an application. In the presence of false negatives, the tool is said to be incomplete with respect to the real set of defects in the *software* (3.19) under analysis. False negatives can be due to several reasons such as 1) the use of heuristics for detecting defects, 2) too restrictive analysis data.

3.9
false positive
observed *defect* (3.4) which does not correspond to a true defect

Note 1 to entry: The term is used for analysis tools producing defect information during the analysis of an application. False positives appear during the analysis because of several possible reasons, such as lack of precision of the analysis rules.

3.10
formal verification
activity proving or disproving the correctness of intended applications with respect to a formal specification or a property, using formal methods of mathematics

3.11
malfunction
behaviour of a system or component that deviates from the specifications

3.12
protection
process to secure content

[SOURCE: ISO/IEC 15444-8:2007, 3.24]

3.13
risk
effect of uncertainty on objectives

Note 1 to entry: ISO 22538-4 defines risk as “probability of loss or injury from a hazard”.

[SOURCE: ISO 31000:2018, 3.1, modified — Notes 1, 2 and 3 to entry have been removed; a new Note 1 to entry has been added.]

3.14
safety
freedom from unacceptable *risk* (3.13)

3.15
safety-critical system
system whose failure or *malfunction* (3.11) may result in one (or more) of the following outcomes:

— death or serious injury to people

— loss or severe damage to equipment/property

— environmental harm

EXAMPLE Examples of safety-critical systems are critical infra-structures, medical equipment, transportation, and nuclear power plants. Safety-critical systems are also sometimes called life-critical systems.

3.16
security
resistance to intentional, unauthorized act(s) designed to cause harm or damage to a system

3.17
security level
combination of a hierarchical *security* (3.16) classification and a security category that represents the sensitivity of an object or the security clearance of an individual

Note 1 to entry: The minimum security level is an indication of the minimum *protection* (3.12) required.

3.18
semi-formal verification
method that is based on a description given in semi-formal notation

3.19
software
all or part of the programs, procedures, rules, and associated documentation of an information processing system

[SOURCE: ISO/IEC 19770-3:2016, 3.1.26, modified — Note 1 to entry has been removed.]

3.20
software item
identifiable part of a *software* (3.19) product, consisting of source code, object code, control code, control data, or a collection of these

Note 1 to entry: Software item is a generic term that designates well-identified parts of software source code, object code or data. A software item belongs to a syntactic category of the programming language in which the software is written. Examples are classes, variables, functions and types. A software item is an identifiable part of a software product.

3.21
software safety
ability of *software* (3.19) to be free from unacceptable *risk* (3.13)

Note 1 to entry: It is the ability of software to resist failure and *malfunctions* (3.11) that can lead to death or serious injury to people, loss or severe damage to property, or severe environmental harm.

Note 2 to entry: Software quality, including software safety, is achieved using software engineering. Software engineering for *safety-critical systems* (3.15) emphasizes the following directions:

— process engineering and management;

— selecting the appropriate tools and environment for the system; the principle of using the best tools fit to the purpose prevails as in most engineering disciplines;

— adherence to requirements.

3.22
software security
ability of *software* (3.19) to protect its assets from a malicious attacker

Note 1 to entry: According to software product quality model in ISO/IEC 25010, software security applies to software assets and is decomposed into the following set of properties: confidentiality, integrity, availability, authentication, authorization and non-repudiation.

3.23
software unit
smallest independent piece of *software* (3.19), which can be independently translated, and which can be tested with the relevant data on whether it performs to specification

3.24
static program analysis
sub-field of formal methods concerned by analyzing the properties of a *software* (3.19) code without executing this code in the target (binary) format

3.25
system safety
ability of a system to be free from unacceptable *risk* (3.13)

Note 1 to entry: A system is defined as a set or group of interacting, interrelated or interdependent elements or parts, that are organized and integrated to form a collective unity or a unified whole, to achieve a common objective. In a broader view the definition of a system consists in the hardware, *software* (3.19), human systems integration, procedures and training. Therefore, system safety is part of the systems engineering process and should systematically address all of these domains and areas in engineering and operations in a concerted fashion to prevent, eliminate and control hazards.

Note 2 to entry: A system safety concept helps the system designer(s) to model, analyze, gain awareness about, understand and eliminate the hazards, and apply controls to achieve an acceptable level of *safety* (3.14). The systems-based approach to safety requires the application of scientific, technical and managerial skills to hazard identification, hazard analysis, and elimination, control, or management of hazards throughout the life-cycle of a system, program, project or an activity or a product. Hazop is one of several techniques available for the identification of hazards.

3.26
target of verification
TOV
software (3.19), or a set of *software items* (3.20) or *units* (3.23), to be verified (e.g. in terms of *safety* (3.14) and *security* (3.16))

Note 1 to entry: Target of evaluation (TOE) is a commonly used term in systems security techniques. TOE is defined as a set of software, firmware and/or hardware possibly accompanied by guidance.

3.27
target software
final product of a *software* (3.19) development process, containing at least the binary code able to run on the target computer

Note 1 to entry: Target software may consist of several files, including binary and source files, libraries, installation and compilation script files, documentation and data files. The target software often relies on underlying layers of software, that are not part of the target software, but that are necessary to be executed on the target platform, for instance libraries.

3.28
target system
complete computing platform capable of running the *target software* (3.27)

Note 1 to entry: A target system consists of hardware resources and *software* (3.19) resources installed on the hardware.

3.29
toolbox
set of tools completing each other in terms of capabilities, to cover a larger area of their intended use

3.30
trust
degree to which a user or other stakeholder has confidence that a product or system will behave as intended

[SOURCE: ISO/IEC 25010:2011, 4.1.3.2]

3.31
use case
specification of a sequence of actions, including variants, that a system (or other *entity* (3.6)) can perform, interacting with actors of the system

[SOURCE: ISO 15745-1:2003, 3.35, modified — The domain "<class>" at the beginning and "[UML]" at the end of the definition has been removed.]

3.32
validation
confirmation, through the provision of objective evidence, that the requirements for a specific intended use or application have been fulfilled

EXAMPLE *Safety* (3.14) validation has been defined as an assurance, based on examination and tests, that the safety goals are sufficient and have been achieved (ISO 26262-1).

[SOURCE: ISO/IEC 25000:2014, 4.41, modified — Note 1 to entry have been removed; EXAMPLE has been added.]

3.33
verification
confirmation, through the provision of objective evidence, that specified requirements have been fulfilled

[SOURCE: ISO/IEC 25000:2014, 4.43, modified — Note 1 to entry have been removed.]

3.34
verification method
method for producing objective evidence that specified requirements of a system have been fulfilled

3.35
verification tool
instrument that can be used during *verification* (3.33) to collect information about the *target of verification* (3.26), to perform interpretation of information or to automate part of the verification

3.36
vulnerability
potential flaw or weakness in *software* (3.19) design or implementation that could be exercised (accidentally triggered or intentionally exploited) and result in harm to the system

Note 1 to entry: The CVE classification (see Reference [25]) defines the de-facto standard classes of the known software vulnerabilities.

Note 2 to entry: A vulnerability is exploitable if it can be activated in practice.

4 Abbreviated terms

AASPE	automated assurance of security policy enforcement tool
ACSL	ANSI/ISO C specification language
ARM	advanced RISC machine
BDD	binary decision diagram
BNF	Backus–Naur form
CASE	computer-aided software engineering tool
CC	common criteria
CSPN	Certification de Sécurité de Premier Niveau
CTL	computation tree logic
dns	domain name server, domain name system
DREAD	damage, reproducibility, exploitability, affected users, and discoverability risk assessment model
EAL	evaluation assurance level
FMEA	failure mode and effects analysis
HAZOP	hazard and operability analysis
LTL	linear temporal logic
OSINT	open-source intelligence
RM	reference manual
SAT	satisfiability
SCL	Safety Culture Ladder
SDLC	software development life-cycle
SMT	satisfiability modulo theory
SQL	structured query language
STRIDE	spoofing, tampering, repudiation, information disclosure, denial of service and elevation of privilege
SWOT	strengths, weaknesses, opportunities and threats
UML	unified modeling language
VDM-SL	Vienna development method - specification language
V&V	verification and validation
XSS	cross-site scripting

5 Models for software safety and security verification tools

The model for software safety and security verification is needed to define capabilities of and requirements for tools by their input, processes and output. The model consists of the following elements:

a) a set of use cases for concretizing the use of software safety and security verification tools,

b) a set of entities that represents identifiable information which appears in software safety and security verification activities, and

c) a categorization for software safety and security verification tools.

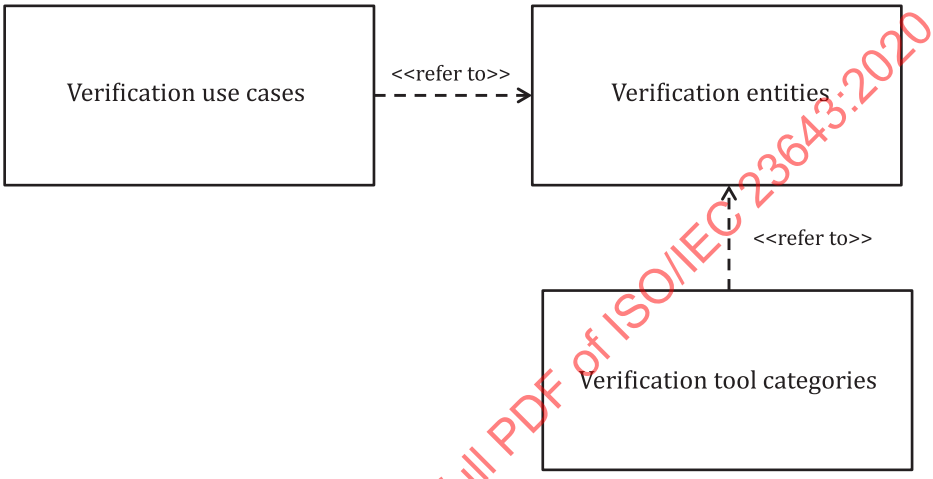

Figure 1 — Overall structure of models of software safety and security verification

Software safety and security verification activities, including use of tools, are needed whenever any person (or a thing) develops safety and/or security critical software. Any change to the software can also require new verification activities (e.g. a whole new verification process). Other typical uses for software safety and security verification tools are related to software quality evaluation and software certification cases. The use of such tools may be part of system or application security verification process, as defined in ISO/IEC 27034 (all parts).

As safety and security issues are becoming increasingly important within software intensive industry and software intensive systems, a lot of new terms and concepts emerge more and more frequently. Terms like cyber security, systems of systems, and Internet of Things were never heard of some years ago, and today they are supposed to be recognized and understood by anyone using or developing software and systems. Unfortunately, there is still a lot of space for loose communication and misunderstanding. Commonly accepted and mutually understood terminology between the developers and users of safety and security verification tools is needed. The most often needed terms and entities are defined in Clause 3 and discussed in Clause 7.

Clause 6 introduces the use cases of software safety and security verification through some usage scenarios. They describe the way how software developers or evaluators are intended to use verification and validation tools to ensure that the end-product becomes of the adequate quality in terms of safety and security. The target level of safety and security is often defined during the requirements stage of each development case.

Both safety and security of software are equally essential. However, there are no single tools that can manage both of such areas of requirements perfectly. That is why the users need to use several tools, representing several tool categories, when they want to convince themselves and other stakeholders to trust the overall safety and security of the target software. Clause 8 introduces the verification tool categories, each specified by capabilities of the tools in the category, and specifies requirements for software safety and security verification tools.

Software industry recognizes several different software development life-cycle (SDLC) models and development approaches, known for example as V-shaped, linear, prototypical or agile. In safety and security critical cases, the work often follows a standard SDLC where the activities of the development process are represented and well-identified with the same names and same order. Software cannot be tested before being programmed. Verification cannot be done and verification tools cannot be used if there are no specified requirements. To highlight the verification opportunities during a standard SDLC, Figure 2 introduces a case of V-shaped life-cycle for developing safety and security critical software.

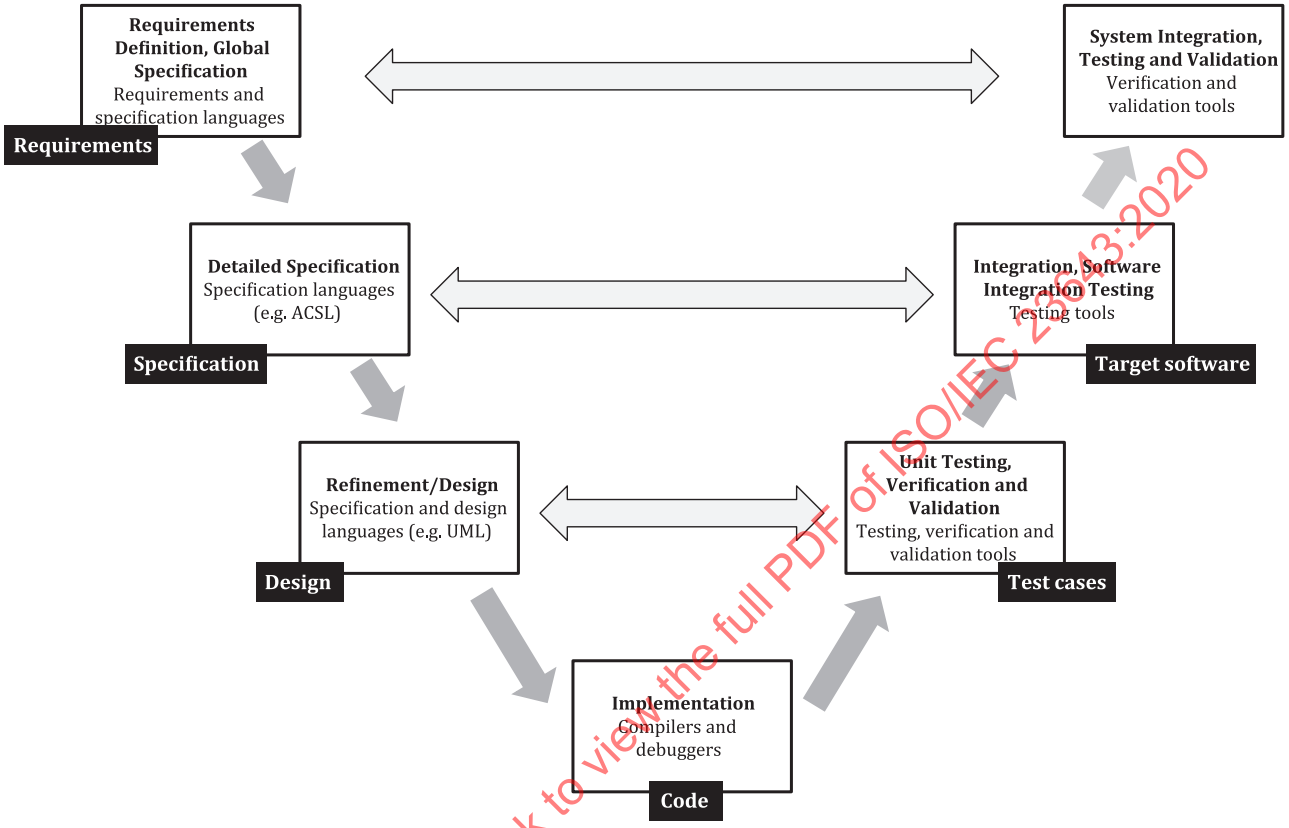

Figure 2 — A simplified view of safety and security critical software development life-cycle

Each box in Figure 2 represents an activity and grey arrows represent the flow of artefacts between the activities. For each activity the text in the box indicates some exemplary tools needed to perform verification and/or validation tasks. The horizontal arrows in Figure 2 represent the links between pairs of activities when V&V activities occur. For each V&V activity on the right side the arrows show the correspondence to a development activity on the left side against which outcomes verification and validation are performed. The related verification actions are actually performed immediately after provision of development outcomes on the left side, before moving to the next development activity, and the validation actions only after the implementation and preceding V&V activities. In more details, the following correspondences occur:

— conformance of unitary V&V to the detailed specifications and to the design;

— conformance of software integration V&V to the specifications;

— conformance of system integration to the requirements and global specifications.

For each activity, Figure 2 indicates its main outcome (also sometimes called main artefact or by-product) as a small black box. Different techniques, methods and tools can be used to provide understandable form for artefacts (e.g. UML, ACSL).

6 Use cases of software safety and security verification tools

6.1 General

By definition the use cases specify a sequence of actions that a system can perform, when interacting with its users (i.e. the actors of the system). In this document, the use cases are defined at a level where the users interact with one or more software safety and security verification tools. Selection of the tools to be used can be made following the processes of ISO/IEC 20741, which introduces the generic process for evaluating and selecting any software engineering tools (see Annex B).

In this document the detailed variations for using specific tools are ignored because the purpose of introducing use cases here is not to give specifications for building the tools but to help the readers to understand the environment where the tools are used. For that purpose, the use cases are not presented in any standard format, but rather as usage scenarios, although the term "use case" is still used.

Actors of software verification are as follows:

— Developers perform the development and verification tasks given in the life-cycle. They produce all intermediate products to be verified. They also perform the validation of the intermediate products and testing, sometimes together with the client.

— Evaluators are verifying and validating the software against some safety or security standard. They are independent from the developers' tasks, but may belong to the same (1st party) or an independent third-party organization.

— Certification bodies examine the verification results already done, perform further verifications if needed and deliver, if acceptable, a certificate to the software verified. Certification bodies are always third-party organizations (ISO/IEC 17000).

Figure 3 introduces the overview of verification use cases and the actors related to them. Purpose of the verification and the criticality of the TOV are the main differentiators of the use cases. The higher the criticality of the TOV is in terms of safety and security, the more demanding the verification process is, and the more expertise is required from the evaluator(s).

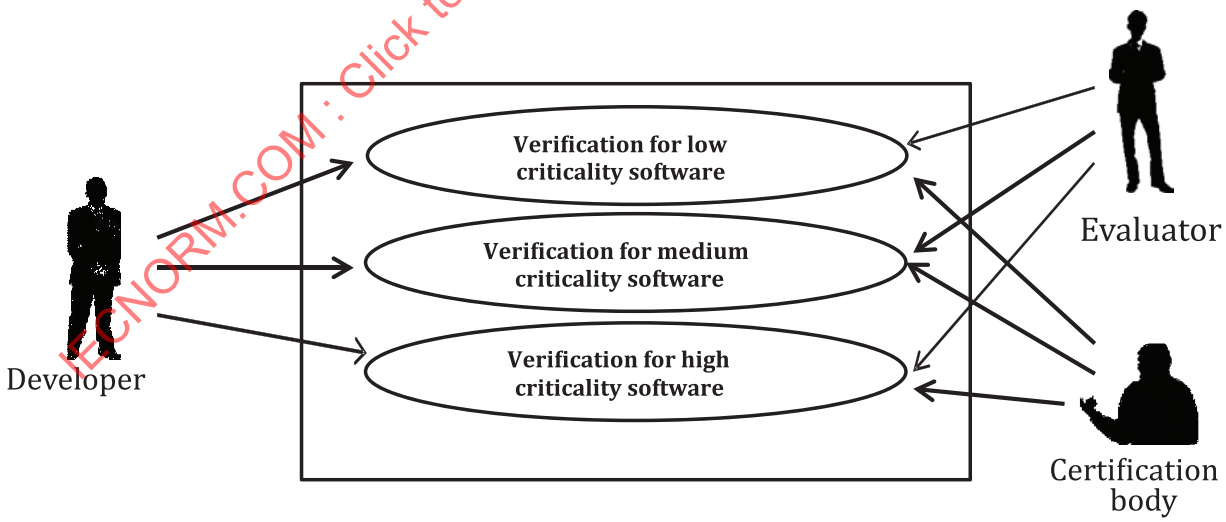

Figure 3 — Use cases of software safety and security verification tools

Formality of the verification varies by the criticality of the TOV. In some cases it is good enough to run an informal or a semi-formal verification whereas in the most critical cases, a formal verification is required. For the lowest criticality level, when the purpose of verification is not to achieve a certificate, an independent evaluator is not needed at all. All verification use cases aiming to certify a TOV shall specify the target safety and/or security level within an applicable standard (e.g. evaluation assurance levels EAL1 to EAL7 of common criteria for information technology security evaluation in

ISO/IEC 15408 (all parts) or DO-178 B/C). The definitions of evaluation assurance levels of Common Criteria are in Annex A.

6.2 to 6.4 introduce the lists of actions of the verification use cases.

6.2 Verification for low criticality software

For low-criticality software, requirements can be handled through informal methods and developers often perform their own testing. If the goal of the verification is certification there are additional actions for reaching a minimum level of safety and security. These additional actions are the optional items e), f), g), h) and k) below.

The scenario may include the following actions:

a) informal specification and design;

b) implementation;

c) verification by testing;

d) integration;

e) selection of an evaluation and certification scheme capable of assessing low criticality software (e.g. CSPN, SCL or CC) — third-party verification;

f) selection of the desired minimum certification level — third-party verification;

g) evaluation of the TOV to a defined level of safety or security — third-party verification;

h) certification by some certification body to obtain approval — third-party verification;

i) deployment/acceptance;

j) update/maintenance;

k) re-evaluation — third-party verification.

6.3 Verification for medium criticality software

Medium criticality developments do not need verification to their full extent, but a rigorous approach with intermediate products (artefacts) is used. The intermediate products may use different languages, which may require annual translations between activities. For each activity it is best to use only one language. If the goal of the verification is certification there are additional steps for reaching a minimum level of safety and security. These additional actions are the optional items c), k), m), n) and q) below.

The scenario may include the following actions:

a) qualification of tools (e.g. compilers);

b) requirements definition using preferably a semi-formal specification language;

c) selection of the certification level and definition of the TOV — third-party verification;

d) detailed specifications using preferably a formal specification language;

e) detailed design, using a semi-formal design language (e.g. UML);

f) implementation;

g) unit testing to give an intuitive understanding of the TOV made so far; the generation of the corresponding test cases can use the specifications to delimit them; testing consists of e.g. in coverage and boundary testing;

h) verification of the components of the TOV, to check the satisfiability of the specifications, e.g. using the later as assertions;

i) integration of the components;

j) integration testing;

k) verification and validation of the integrated software — third-party verification;

l) system integration;

m) evaluation of the TOV to a defined level of safety or security — third-party verification;

n) certification to obtain approval — third-party verification;

o) deployment/acceptance;

p) update/maintenance;

q) re-evaluation — third-party verification.

6.4 Verification for high criticality software

Any software provider organization developing high criticality components (e.g. embedded software) can be willing to verify the safety and security of their products first without certification and start the official third-party evaluation later.

This use case deals with the production of high criticality software. The scenario may include the following actions:

a) qualification of tools (e.g. compilers);

b) requirements definition using preferably a semi-formal specification language;

c) selection of the certification level and definition of the TOV — third-party verification;

d) detailed specifications using a formal specification language (e.g. VDM-SL, B, Z, or ACSL);

e) specifications refinement (optional) or formal design with a formal compliance check;

f) implementation;

g) unit testing to give an intuitive understanding of the TOV made so far;

h) completing unit testing by static program analysis — third-party verification;

i) verification of the components of the TOV, to extract low-level (e.g. run-time) faults and high-level behavioural faults; the later checks the satisfiability of the specifications by the code; the formal specifications produced above are used for compliance verification, using e.g. Hoare Logic and formal proofs are done;

j) formal verification of the components — third-party verification;

k) integration of the components;

l) integration testing;

m) verification and validation of the integrated TOV;

n) system integration;

o) system testing on the target system;

p) evaluation of the TOV to a defined level of safety or security — third-party verification;

q) deployment/acceptance;

r) update/maintenance;

s) re-evaluation — third-party verification.

7 Entity relationship chart of software safety and security verification

It is important to use common and well-defined terms with software safety and security verification tools, services, and methods, that might be evaluated, compared and selected by any third-party representatives. The nature of and the relationships among the most important entities can be understood in similar ways by all the actors of a verification activity.

The entities involved in the area of interest of this document are organized in the diagram in Figure 4.

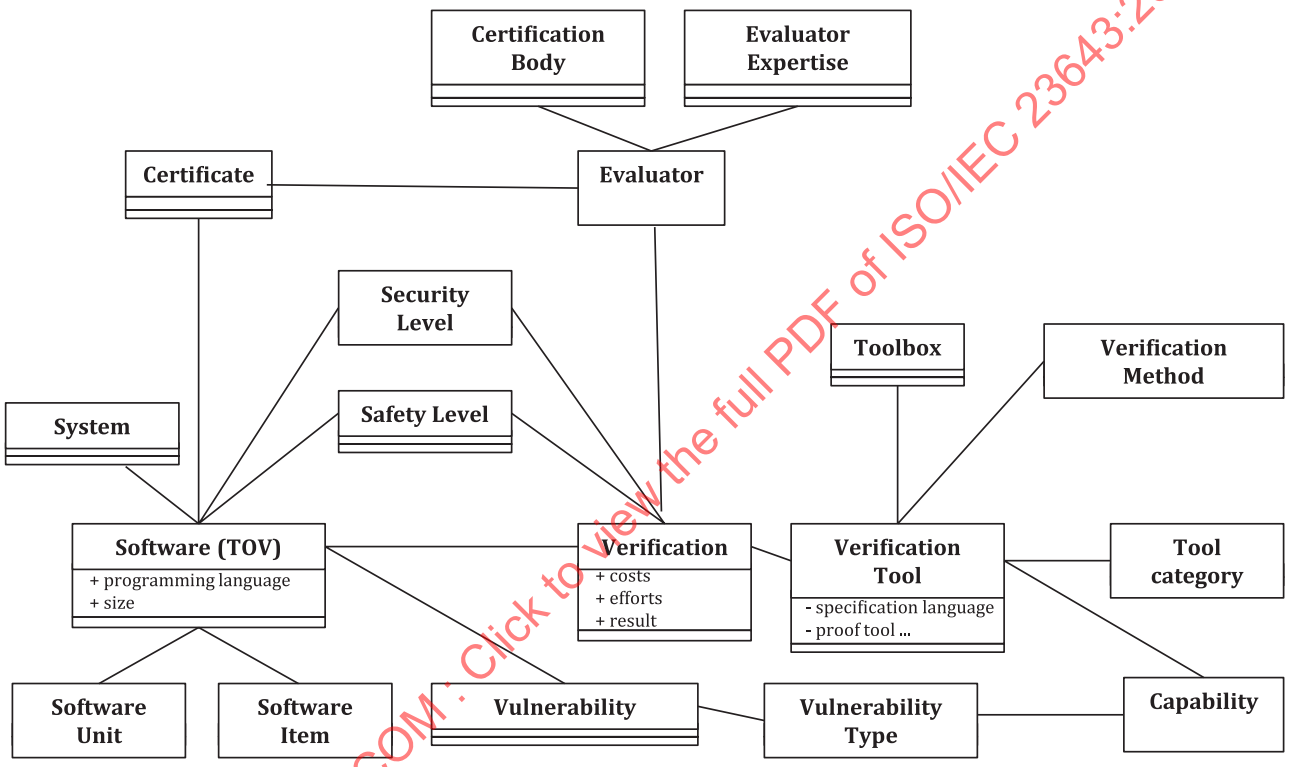

Figure 4 — Entity relationship chart of software safety and security verification

The most important entities in the world of software safety and security verification tools are “software”, “verification” and “verification tool”. In every instance of verification, the target of verification means software, which consists of one or more software units and/or software items. The software may be part of a system or several systems, but it is not relevant from the software verification point of view.

A person called evaluator is running the verification, where he or she uses one or more verification tools. The tools are usually based on one or more verification methods (e.g. dynamic or static program analysis or formal verification). Tools based on same methods often represent the same tool category, having similar capabilities for identifying potential vulnerabilities of the software (i.e. TOV). Several tools with different capabilities that mostly complete each other, may together constitute a toolbox. Selection of an applicable toolbox or set of applicable tools can be made based on the known required safety and/or security levels of the TOV.

The target of verification (software) may be certified by the evaluator representing a certification body, if the verification proves the software free of vulnerabilities at the target safety and security levels. The required expertise of the evaluator may be higher, if the purpose of verification is to achieve a certificate.

8 Categories, capabilities of and requirements for software safety and security verification tools

8.1 General

As software, by definition, consists of all or part of the programs, procedures, rules, and associated documentation of an information processing system, it may be complicated to verify. Because of the complexity of software, there are many kinds of verification tools. In this document the types of software safety and security verification tools are divided into several categories, specified by the capabilities of the tools included in the category. The capabilities enable the tools to analyze and verify some software code or artefacts. Verification activities are human guided, but very often tool assisted, and the level of automation varies category by category, and tool by tool within the category. Verification capabilities allow performing various kinds of verification activities in various tasks of a software development process. Figure 5 introduces the categories of software safety verification tools, and Figure 6 the categories of software security verification tools.

8.2 Categories of software safety verification tools

8.2.1 General

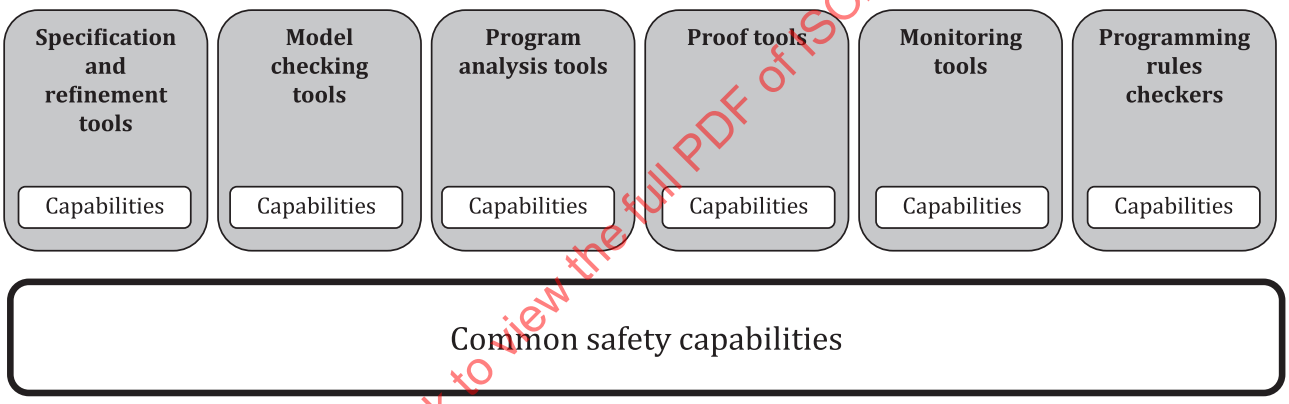

Figure 5 — Categories of software safety verification tools

8.2.2 to 8.2.7 define the categories of software safety verification tools introduced in Figure 5.

8.2.2 Specification and refinement tools

This safety verification tool category encompasses all tools implementing formal specification, i.e. mathematical based specification techniques and languages. They are used to describe a software system, to analyze its behaviour, and to aid in its design by verifying properties of interest by means of rigorous and effective reasoning. These specifications are formal in the sense that they have a syntax, a formal semantics and they can be used to infer useful information.

With a specification it is possible to use the formal verification techniques below to demonstrate that a system design and/or implementation is correct with respect to its specification. Another approach is to refine a specification step by step in a correct manner to transform it into a design, which is then transformed into an implementation that is correct by construction.

8.2.3 Model checking tools

Model checking aims mainly at automatically verifying properties of finite-state systems. This safety verification tool category contains all tools that enable to check automatically or not some specification on a given model (e.g. an automaton) that represents the behaviour of a system. The specification

can contain safety and liveness properties. The model can be extracted from the source code of the application (e.g. the control flow graph) or can be produced independently (e.g. a Petri Net).

8.2.4 Program analysis tools

In the context of this document, dealing with software safety and security verification tools, the safety verification tool category of program analysis tools encompasses tools that can discover faults and vulnerabilities during the programming phases of the development life-cycle. This category can therefore also be called “static program analysis tools”. With this definition, this category can be considered as a generic category that also encompasses some of the other categories listed here, but we distinguish between the static program analysis techniques that apply directly to the source code and the techniques that apply to some other representation of the code and its artefacts, keeping only the former tools for this category.

Due to the various underlying techniques, some of them being computationally undecidable, program analysis tools cannot always terminate with the right answer (yes or no) and may return false negatives, false positives or no answer (because they never terminate).

8.2.5 Proof tools

This safety verification tool category contains all tools aimed at proving mathematically some formal specification by means of some computer program called theorem prover. This category decomposes into two main sub-categories:

— automatic theorem provers, and

— proof assistants also called interactive theorem provers.

8.2.6 Monitoring tools

This safety verification tool category groups all tools that check at runtime the specification of a software application. A specification may deal with computing resources consumption, safety or security properties. Most of these properties are stated using a specification language or a subpart thereof that can be executed or evaluated.

Contrary to the program analysis tools category, monitoring tools may not be exhaustive as they verify some specification on a limited number of executions only. Due to their dynamic nature, the performance of these tools is important.

8.2.7 Programming rules checkers

Programming rules checkers enforce syntax and semantic rules further to those given in the programming language reference manual. For instance, they provide warnings and force the programmers to respect a uniform style, notation, i.e. a set of conventions. Programming rules are sometimes associated to a community of programmers (e.g. in a given project) or associated to a given application domain (e.g. automobile software industry uses the MISRA C/C++ rules).

8.3 Categories of software security verification tools

8.3.1 General

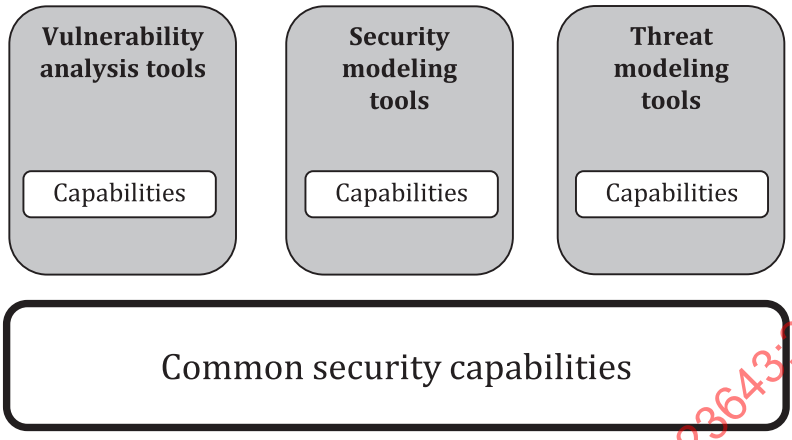

Figure 6 — Categories of software security verification tools

8.3.2 to 8.3.4 define the categories of software security verification tools introduced in Figure 6.

8.3.2 Vulnerability analysis tools

This category contains security verification tools that are capable to discover the attack surface or to find vulnerabilities in TOV. Vulnerability analysis is different from software testing, since vulnerability analysis tries to find a way in which the software should not work. Therefore, negative specifications should be evaluated during a vulnerability analysis (e.g. “only the valid password should be accepted, nothing else”). Because of the nature of vulnerability analysis, the tools in this category generally will not provide evidence of the software being free from a specific type of vulnerabilities but can provide a list of vulnerabilities found.

8.3.3 Security modeling tools

The tools in this security verification tool category help to perform security modeling, such as defining vulnerable assets, security objectives, and security requirements. The results of these modeling tools will be the basis of further verification activities. They aid defining the exact objectives and requirements, which the verification process should confirm.

8.3.4 Threat modeling tools

This category of security verification tools contains all tools supporting threat modeling and analysis techniques, such as attack trees, attack impact, attack surface, fault trees, misuse cases, attacker profiling, and the STRIDE threat classification model. The aim of threat analysis phase is to model how a system should not work. This analysis can use the results of the security modeling tools and as a result specify the threats to be verified during the further steps of verification.

NOTE STRIDE is a model for identifying computer security threats. It provides a mnemonic for security threats in six categories. The threats are: **S**poofing of user identity, **T**ampering, **R**epudiation, **I**nformation disclosure (privacy breach or data leak), **D**enial of service (D.o.S), and **E**levation of privilege. STRIDE was initially created as part of the process of threat modeling used to help reason and find threats to a system.

8.4 Capabilities of software safety and security verification tools

This document specifies the potential capabilities for safety and security verification tools of each category. Table 1 introduces the typical capabilities of software safety verification tools of each related category, and Table 2 the typical capabilities of software security verification tools by their categories.

The vendor of a software safety and security verification tool shall specify the category and capabilities of the tool, so that the users know what they get when acquiring such a tool or a toolbox including several tools.

Table 1 — Capabilities of software safety verification tools by category

Capability list of software safety verification tools		
Tool category	**Capability**	**Remarks**
Specification and refinement tools	Abstract data-types	
	Predicates calculus	
	Algebraic specifications	
	Temporal logics	Relates to model checking
	Functional specifications	
Model checking tools	Symbolic model checking	E.g. BDD, see NOTE 1
	Bounded model checking	
	Explicit-state model checking	
	Partial order reductions	
	Abstractions	
Program analysis tools	Hoare logics	E.g. Frama-C
	Abstract interpretation	E.g. Frama-C
	Tableau calculus	
	Control and data flow graphs analysis	
	Compilers	
Proof tools	Deductive proofs	
	Unification	
	Satisfiability modulo theories	See NOTE 2
	Satisfiability solvers	See NOTE 3
	Proof assistants	E.g. Coq
Monitoring tools	Off-line monitoring	Results are exploited after execution
	Online monitoring	Results are exploited during execution. This category sub-divides into the next two categories
	In-line monitoring	Monitors are built into the code
	Out-line monitoring	Monitors are separated from the code
	Memory debuggers	Used frequently during debugging
	Assertions checkers	Mostly known
	Temporal runtime verifiers	Verify temporal properties

NOTE 1 Binary decision diagrams (BDD) are data structures for representing boolean functions, that is, functions that take booleans as inputs and produce a boolean as output. These data structures can be considered as compressed.

NOTE 2 In computer science, the satisfiability modulo theories (SMT) problem is a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic with equality.

NOTE 3 In computer science, the boolean satisfiability problem (sometimes called propositional satisfiability problem and abbreviated as SATISFIABILITY or SAT) is the problem of determining if there exists an interpretation that satisfies a given boolean formula. Therefore a SAT solver is a software application that solves this problem.

NOTE 4 HAZOP stands for HAZar and OPerability study and is a structured and systematic examination of a complex planned or existing process or operation in order to identify and evaluate problems that may represent risks to personnel or equipment. The intention of performing an HAZOP is to review the design to pick up design and engineering issues that may otherwise not have been found. The technique is based on breaking the overall complex design of the process into a number of simpler sections called 'nodes' which are then individually reviewed. It is carried out by a suitably experienced multi-disciplinary team (HAZOP) during a series of meetings.

Table 1 *(continued)*

Capability list of software safety verification tools		
Tool category	**Capability**	**Remarks**
Programming rules checkers	Syntactic programming rules Semantic programming rules	Style checkers
Common safety capabilities	Parsers	Based on language grammars
	Semantic analysis	Based on language semantics
	Code generation	binary, byte-code, high-level programming language
	Risks analysis	E.g. HAZOP (See NOTE 4), DELPHI, SWOT and FMEA

NOTE 1 Binary decision diagrams (BDD) are data structures for representing boolean functions, that is, functions that take booleans as inputs and produce a boolean as output. These data structures can be considered as compressed.

NOTE 2 In computer science, the satisfiability modulo theories (SMT) problem is a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic with equality.

NOTE 3 In computer science, the boolean satisfiability problem (sometimes called propositional satisfiability problem and abbreviated as SATISFIABILITY or SAT) is the problem of determining if there exists an interpretation that satisfies a given boolean formula. Therefore a SAT solver is a software application that solves this problem.

NOTE 4 HAZOP stands for HAZar and OPerability study and is a structured and systematic examination of a complex planned or existing process or operation in order to identify and evaluate problems that may represent risks to personnel or equipment. The intention of performing an HAZOP is to review the design to pick up design and engineering issues that may otherwise not have been found. The technique is based on breaking the overall complex design of the process into a number of simpler sections called 'nodes' which are then individually reviewed. It is carried out by a suitably experienced multi-disciplinary team (HAZOP) during a series of meetings.

Table 2 — Capabilities of software security verification tools by category

Capability list of software security verification tools		
Tool category	**Capability**	**Remarks**
Vulnerability analysis tools	Network tools	Port scanner, network mapping, network analysis
	OSINT	shodan, etc
	Common enumerators	Brute-forcers, dns vulnerability scanner, etc.
	Code review	
	Disassembler	
	Decompiler	
	Code analysis	
	Configuration analysis	OS, app
	Hardening check	Verification of usage of Stack cookie, ASLR, Control Flow Protection, FlowGuard, etc.
	Known vulnerability check	
	Emulation	
	Symbolic execution	
	Common vulnerability scanner	SQL injection, buffer overflow, XSS
	Performance analysis	
	Monitoring	Information leakage monitor, Monitoring capabilities from safety verification tools
	Log analysis	
	Fuzzing	
Security modeling tools	Security related features of software development tools	Code editors, compilers, linkers, assemblers, disassemblers, debuggers, testing, CASE tools, version control
	Informal walktrough	
	Formal inspection of binary and source code	
Threat modeling tools	Attack trees	E.g. https://github.com/cmu-sei/AASPE
	Attack impact	
	Attack surface	
	Fault trees	
	Misuse cases	
	STRIDE	
	DREAD	See NOTE 1
	System modeling	E.g. UML
	Other modeling technique	https://www.owasp.org/

NOTE 1 DREAD is part of a system for risk-assessing computer security threats previously used at Microsoft that provides a mnemonic for risk rating security threats using five categories. The categories are: Damage — how bad would an attack be? Reproducibility — how easy is it to reproduce the attack? Exploitability — how much work is it to launch the attack? Affected users — how many people will be impacted? Discoverability — how easy is it to discover the threat?

NOTE 2 HAZOP stands for HAZard and OPerability study and is a structured and systematic examination of a complex planned or existing process or operation in order to identify and evaluate problems that may represent risks to personnel or equipment. The intention of performing an HAZOP is to review the design to pick up design and engineering issues that may otherwise not have been found. The technique is based on breaking the overall complex design of the process into a number of simpler sections called 'nodes' which are then individually reviewed. It is carried out by a suitably experienced multi-disciplinary team (HAZOP) during a series of meetings.

NOTE 3 This table provides examples of suitable products available commercially. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO or IEC of these products.

Table 2 *(continued)*

Capability list of software security verification tools		
Tool category	**Capability**	**Remarks**
Common security capabilities	Parsers	Based on language grammars
	Semantic analyzers	Based on language semantics
	Code generation	Binary, byte-code, high-level programming language
	Risks analysis tools	E.g. HAZOP (See NOTE 2), DELPHI, SWOT and FMEA

NOTE 1 DREAD is part of a system for risk-assessing computer security threats previously used at Microsoft that provides a mnemonic for risk rating security threats using five categories. The categories are: Damage — how bad would an attack be? Reproducibility — how easy is it to reproduce the attack? Exploitability — how much work is it to launch the attack? Affected users — how many people will be impacted? Discoverability — how easy is it to discover the threat?

NOTE 2 HAZOP stands for HAZard and OPerability study and is a structured and systematic examination of a complex planned or existing process or operation in order to identify and evaluate problems that may represent risks to personnel or equipment. The intention of performing an HAZOP is to review the design to pick up design and engineering issues that may otherwise not have been found. The technique is based on breaking the overall complex design of the process into a number of simpler sections called 'nodes' which are then individually reviewed. It is carried out by a suitably experienced multi-disciplinary team (HAZOP) during a series of meetings.

NOTE 3 This table provides examples of suitable products available commercially. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO or IEC of these products.

Several, probably the most commonly known, software verification tools address the software source code, written in C, C++ and Java, which are common programming languages within safety and security critical dynamic industrial applications. However, as the number of verification tool categories and lists of their capabilities show, there are many tools addressing other components of software than just the source code (e.g. documentation or object code).

A software safety and security verification tool may have capabilities from more than one tool category, but it shall have capabilities from at least one tool category.

8.5 Common requirements for safety and security verification tools

The requirements for safety and security verification tools may be either common to all tools, or specific to a certain tool category. This subclause provides a list of common requirements and 8.6 to 8.14 give the category specific requirements for each category.

The following requirements are common to all tools:

— The vendor of a safety and security verification tool shall provide information about the tools applicability in terms of safety and security analysis. It shall explain which activities of a development life-cycle are covered by the tool and which safety and/or security properties are handled.

— The vendor of a safety and security verification tool shall provide information about the tools applicability in terms of verification use cases. This allows the user of the tool to check if the tool is adequate to the problem at hand, i.e. to 1) the specific application to be analyzed, 2) the desired safety and security properties, and 3) the environment (e.g. industrial setting and constraints) in which the tool will be put into practice. See NOTE.

— The vendor of a safety and security verification tool shall provide information about the category and capabilities of the tool, as defined in the Tables 1 and 2.

— The vendor of a safety and security verification tool shall provide the tools requirements in terms of execution platform, i.e. computer, operating system and required middleware and libraries.

— The vendor of a safety and security verification tool shall provide information allowing to compare its detection capabilities with other tools from the same category.

NOTE The term industrial setting denotes the industrial environment in which the tool is put to work, such as 1) CASE tool used in house, 2) the in-house development (including validation) process, 3) computing platforms used (e.g. ARM/x86, cross-compilers, OS kind and version), etc.

8.6 Requirements for specification and refinement tools

A safety and security verification tool that works by evaluating specifications and refinements of software is always based on one formal specification language. Therefor the vendor of the tool shall provide:

— a specification language reference manual (RM);

— a grammar (e.g. BNF) for describing the syntax of the language;

— semantics describing the meaning of each construct of the language;

— a specification type-checker, compliant with the RM.

The vendor of a specification tool shall provide information about the tools specification language expressivity and applicability. An example of such requirement is a functional specifications tool precising the order (e.g. predicates, first or higher) and nature of the logics underlying the specification language, e.g. CTL (Computation Tree Logic), LTL (Linear Temporal Logic) or separation logic.

If the specification tool permits to use predefined libraries (i.e. mathematical functions), then the vendor of the tool shall provide the specifications of each supported library. These specifications shall be provided completely in the form of files associated to the tool, in a readable format, and shall be introduced in the RM (e.g. structure of the specifications and location in terms of predefined files).

If the specification tool allows specification refinements, then the vendor of the tool shall provide rules for writing and checking refinements.

If the specification language of a specification tool is bound to a specific programming language, then the vendor of the tool shall provide the references to the programming language, including its language RM. The vendor of a specification tool shall provide the programming language coverage and indicate what syntactical parts of the later are supported and unsupported.

8.7 Requirements for model checking tools

Model checking tools operate on the source code as being the model or on another representation.

When a model checking tool operates on the source code as being the model, the vendor of the tool shall provide the following technical elements:

— a model checking tool user manual (UM);

— a reference to the specification language used to express properties, if necessary;

— benchmarks, defining in which applications the tool has been used and the performance observed (analysis time, number and nature of the faults obtained);

— if the analysis tool supports a specific programming language, then the references to the programming language, including its language RM;

— the programming language coverage, indicating what parts of it are supported and unsupported.

When a model checking tool operates on another representation, the vendor of the tool shall provide the following technical elements:

— the model representation language (e.g. timed automata) with syntax and semantics;

— a methodology for building models in the model representation language;

— detailed traces when the model checking tool fails to verify some property;

— a methodology to check the conformance between models and the underlying programming language, if necessary.

8.8 Requirements for program analysis tools

The vendor of a safety and security verification tool that belongs to the category of program analysis tools, shall provide information about the programming language(s) addressed by the tool as well as the type of the verification technique underlying it (e.g. Hoare logics, abstract interpretation).

The vendor of the tool shall provide the following technical elements:

— an analysis tool UM;

— a reference to the specification language used to express properties, if necessary;

— a program analysis tool;

— benchmarks, defining which applications the tool has been used and the performance observed (analysis time, number and nature of the faults obtained);

— capability to differentiate the target and host platforms, e.g. to allow analyzing a code running for a target computing platform different from the analysis platform.

If a program analysis tool supports a specific programming language, then the vendor of the tool shall provide the references to the programming language, including its language RM. For instance, if the tool is an abstract interpreter, and is therefore tightly bound to a specific programming language, then the references to the latter shall be provided.

Furthermore, the vendor of a program analysis tool shall provide the programming language coverage and indicate which syntactical parts of the language are supported and which are unsupported. For the unsupported syntactical parts, the program analysis tool's manual should include guidance on how these parts could be handled. If the program analysis tool permits to use predefined libraries (e.g. mathematical functions) of the associated programming language, then the vendor of the tool shall provide the method used to analyze each library supported. The built-ins of a program analysis tool shall be fully described in the UM.

If a program analysis tool requires the use of proof tools, then the references to these tools shall be provided.

8.9 Requirements for proof tools

As the category contains tools proving mathematically formal specifications, the vendors of the tools shall provide adequate information about:

— the mode: automated or manual tool (e.g. proof assistant);

— the kind of reasoning used: inference based, SAT solver, resolution based, etc.;

— the logic/language used for expressing predicates: first order logic, higher order logic, typed/untyped, etc.;

— the use of strategies and tactics to guide proofs; language for defining new tactics and strategies;

— the list of possible results of a proof, e.g. true, false, fail, unknown, etc.;

— the possibility to obtain a counter-example in the case of a failing proof;

— benchmarks, defining for which applications the proof tool has been used and the performance observed (execution time, result obtained).

8.10 Requirements for monitoring tools

The vendor of a software safety and security verification tool that belongs to the category of monitoring tools shall describe:

— the nature of the supported programming language (e.g. C/C++);

— the type of monitored properties;

— the specification language to express the desired properties, if necessary;

— the limitations in terms of programming language constructs that are supported (e.g. multi-threading and assembly code);

— the capability to provide the user with a counter-example when a property is not satisfied and the counter-example format (e.g. format of an execution trace);

— execution time and memory consumption rates for comparing the execution of a program with or without monitoring;

— whether monitored properties are state-based or trace-based;

— the invasiveness of the tool in terms of execution environment disturbance (e.g. control flow change, memory layout modification and program object structure modification).

8.11 Requirements for programming rules checking tools

A programming rules checker works at the syntactic and/or semantic level of some programming language. The vendor of a programming rules checker shall provide information about:

— the target programming language for which rules are checked;

— the rules enforced or a reference to a programming rules standard;

— the target programming language standards and variants considered;

— the tool's results shall refer to the programming language RM whenever possible, so that in the case of non-compliance users can understand precisely its cause;

— ease of configuration of the tool (e.g. select a sub-set of rules for checking);

— nature and structure of provided statistics;

— if possible, integration with code analysers.

8.12 Requirements for vulnerability analysis tools

The vendor of a vulnerability analysis tool shall provide information about:

— supported analysis technique;

— type of the vulnerability analysis (information gathering, static program analysis, dynamic program analysis);

— attack surfaces discovered and scanned, and the types of vulnerabilities that can potentially be discovered (preferably, referencing prerequisites for successful identification as well);

— what kind of security requirements the tool can check against, and with what level of confidence;

— inventory of the vulnerabilities found during the analysis.

8.13 Requirements for security modeling tools

The security modeling process starts with defining the scope by specifying what is covered and what is not covered in the verification. After specifying the verified systems and applications the security modeling generally requires collecting data from various sources.

The vendor of a security modeling tool shall provide information about:

— supported modeling techniques;

— inventory of the supplied modeling properties that are supported, such as assets, security objectives or security requirements;

— exporting model format(s);

— specification of the model format, as this output shall be used in further verification activities;

— support verifying the model against completeness or minimum requirements (e.g. the tool shall verify whether every asset is covered with at least one security objective, or whether every security objective is protected with at least one security requirement).

8.14 Requirements for threat modeling tools

The vendors of threat modeling tools shall provide information about:

— supported threat analysis and modeling techniques;

— inventory of the collected threats;

— properties of collected threats, including the description of the threat, reference to the affected asset and relation to the security objective or to other security elements.

Annex A
(informative)

Evaluation assurance levels of ISO/IEC 15408 common criteria

A.1 General

The evaluation assurance levels (EALs) provide an increasing scale that balances the level of assurance obtained with the cost and feasibility of acquiring that degree of assurance. The ISO/IEC 15408 approach identifies the separate concepts of assurance in a target of evaluation (TOE) at the end of the evaluation, and of maintenance of that assurance during the operational use of the TOE.

A.2 Evaluation assurance level 1 (EAL1) — Functionally tested

EAL1 is applicable where some confidence in correct operation is required, but the threats to security are not viewed as serious. It will be of value where independent assurance is required to support the contention that due care has been exercised with respect to the protection of personal or similar information.

EAL1 requires only a limited security target. It is sufficient to simply state the security functional requirements (SFR) that the TOE must meet, rather than deriving them from threats, organizational security policies and assumptions through security objectives.

EAL1 provides an evaluation of the TOE as made available to the customer, including independent testing against a specification, and an examination of the guidance documentation provided. It is intended that an EAL1 evaluation could be successfully conducted without assistance from the developer of the TOE and for minimal outlay. An evaluation at this level should provide evidence that the TOE functions in a manner consistent with its documentation.

EAL1 provides a basic level of assurance by a limited security target and an analysis of the SFRs in that security target using a functional and interface specification and guidance documentation, to understand the security behaviour.

The analysis is supported by a search for potential vulnerabilities in the public domain and independent testing (functional and penetration) of the TOE security functionality.

EAL1 also provides assurance through unique identification of the TOE and of the relevant evaluation documents.

This EAL provides a meaningful increase in assurance over unevaluated IT.

A.3 Evaluation assurance level 2 (EAL2) — Structurally tested

EAL2 requires the co-operation of the developer in terms of the delivery of design information and test results, but should not demand more effort on the part of the developer than is consistent with good commercial practice. As such it should not require a substantially increased investment of cost or time.

EAL2 is therefore applicable in those circumstances where developers or users require a low to moderate level of independently assured security in the absence of ready availability of the complete development record. Such a situation may arise when securing legacy systems, or where access to the developer may be limited.