*Note to the Reader:* This document was created for the purpose of supporting the 2nd Workshop on Theorem Proving in Certification to be held in Fall 2011.

*Although described as a "standard", the contents of this document are not authorized, approved or recognized by any standards body or any other organization. For the sake of clarity, this document should be described as a "prototypical standard".*

*The certification objectives expressed by this document are intended exclusively for the purpose of understanding different approaches to certification using theorem proving. Neither Critical Systems Labs Inc. nor the organizers of this workshop recommend the use of these objectives as a basis for the certification of any real system.*

# The CSL/Tamarack Certification Standard for Critical Software

Version 1.0, July 2011

This certification standard provides a set of 24 objectives that can be used to assess a software system. These objectives are intended to serve as a basis for evaluating the extent to which the software system is a correct and robust implementation of a set of system requirements.

While the terminology and underlying concept of this standard are influenced by RTCA DO 178, this standard is not intended to be a summary or simplification of DO 178 or any other guidance.

The scope of this standard is limited to activities that begin after a set of system requirements have been specified and allocated to software. The activities that contributed to the specification of the system requirement are outside the scope of this standard. Similarly, the adequacy of the system requirements and the activities performed to validate the software system with respect to the system requirements is outside the scope of this standard.

This certification standard is based on a generalized lifecycle model that consists of the following five phases:

1. Development of the software requirements
2. Development of the software architecture
3. Development of the source code
4. Generation of the executable object code
5. Verification of the executable object code.

This standard recognizes that there are variants of this generalized lifecycle model that are also acceptable for the development and verification of software, e.g., iterative development.

The term source code should be broadly interpreted as reference to all artifacts from which the executable object code is directly generated, or that directly affect the generation of executable object code including:

1. Source code written in a general-purpose programming language

2. Models from which source code or executable object code is generated
3. Configuration data or adaptation data from which source code or executable object code is generated
4. Scripts that guide or control the generation of source code or executable object code.

A software system complies with this standard if the following 24 objectives have been achieved. The numbers shown in brackets at the end of the name of each objective are references to notes that appear at the end of this standard. It is possible to use formal methods to partially or fully satisfy most of the following 24 objectives.

A. Certification is planned [1,2]
B. Tools are qualified [3,4,5]
C. Software requirements are complete, consistent, unambiguous and verifiable
D. Software requirements comply with the system requirements [6,7]
E. Software requirements are traceable to the system requirements [8,9]
F. Software requirements are compatible with the target computer [10]
G. Software architecture is defined
H. Software architecture complies with the software requirements [11]
I. Software architecture achieves sufficient partitioning to protect critical functions
J. Software architecture is compatible with the target computer [12]
K. Source code complies with the software requirements [13,14]
L. Source code is traceable to the software requirements [15]
M. Source code is accurate [16]
N. Source code complies with standards
O. Source code is compatible with the target computer [17]
P. Output of software integration process is complete and correct [18]
Q. Executable object code complies with the software requirements
R. Executable object code is robust with the software requirements [19]
S. Executable object code is compatible with the target computer
T. Verification coverage of the software requirements is achieved [20]
U. Verification coverage of the software structure is achieved [21,22]
V. Verification procedures are correct [23]
W. Verification results of the executable object code are traceable to requirements [24]
X. Verification results are correct and discrepancies resolved [25]

Notes:

1. There must be a certification plan that identifies specific activities that are intended to yield results that will be included in the certification data submitted to the certification authority.

2. When a formal method is used to produce any part of the certification data, the certification plan must identify those activities that involve the use of a formal method. When a formal method is used in combination with other methods, the certification plan must explain how the combination

will yield a complete set of results, i.e., no less complete than what would have been achieved by a single method. This includes an explanation of how the combined approach avoids the possibility of "gaps" between what is covered by the use of a formal methods and what is covered by the use of other methods.

3. With respect to tool qualification, a formal method is a tool that must be qualified. The qualification of a formal method should be broadly interpreted to include the qualification of all elements of the formal method including the formalism(s), software tool(s) and any non-standard axioms, assumptions or abstractions that are "built into" the formal method.

4. As part of the qualification of a formal method, the formal method should be precisely identified to the extent that all activities performed using the formal method could be independently duplicated by a third party.

5. As part of the qualification of a formal method, the soundness of all elements of the formal method should be demonstrated.

6. Compliance of the software requirements with the system requirements can be demonstrated by showing that any implementation of the system that satisfies the software requirements will also satisfy the system requirements.

7. A formal method may be used to show that the software requirements comply with the system requirements by proving that a formal model of the system requirements allocated to software is logically implied by a formal model of the software requirements, on the condition that the formal model of the software requirements is shown to be a conservative representation of the software requirements.

8. Traceability of the software requirements can be demonstrated by showing that all system requirements have been developed into software requirements and that all software requirements are either derived requirements or necessary to fully capture the intent of the system requirements.

9. In the context of this certification standard, the term "derived requirement" refers to a software requirement which is not traceable to a system requirement, but instead, expresses a design decision at the software level that needs to be flowed up to system/software interface.

10. Showing that the software requirements are compatible with the target computer might, for example, involve showing that non-functional requirements such as real-time performance and capacity can be realized on the target hardware.

11. Showing that the software architecture complies with the software requirements can be demonstrated by showing that it is feasible to implement the software requirements, e.g., the

software architecture provides a means of scheduling a process to meet hard real-time deadlines if the process implements hard real-time requirements.

12. Showing that the software architecture is compatible with the target computer might, for example, involve showing that mechanisms of the architecture that depend on hardware can be implemented on the target computer, e.g., hardware interrupts.

13. A formal method may be used to show that the source code complies with the software requirements by proving that a formal model of the software requirements is logically implied by a formal model of the source code, on the condition that the formal model of the source code is shown to be a conservative representation of the source code.

14. A formal model of the source code might be the result of applying a formal semantics to the actual source code, providing it can be shown that the formal semantics is sound.

15. Traceability of the source code can be demonstrated by showing that all software requirements have been developed into source code and that all of the source code is necessary to fully implement the software requirements.

16. The objective of showing that the source code is accurate involves showing that the source code avoids the use of language features or patterns that could undermine the extent to which the actual execution of the software will behave in a manner that can be accurately predicted by the source code. This should also involve the consideration of stack usage, memory usage, fixed point arithmetic overflow and resolution, resource contention and limitations, worst-case execution timing, exception handling, use of uninitialized variables, unused variables or constants, and data corruption due to task or interrupt conflicts. This should also consider uncertainties that may be due to so-called unsafe language features that may yield different behaviors depending on the compiler (including options), linker (including options) and target hardware.

17. Showing that the source code is compatible with the target computer might, for example, involve showing the source code only uses hardware-dependent features of the programming language in a manner that is consistent with the target hardware.

18. Showing that the output of software integration process is complete and correct is a matter of showing that the result of all steps in the process of generating executable object code from source files preserves the behavior expressed by the source files. In a conventional approach, confirmation of this objective is usually achieved by means of review and analysis. Alternatively, a formal method may be used to partially or fully show that this objective has been achieved, e.g., by means of a formal proof of correctness for the compiler or by means of formal de-compilation process.

19. Robustness is the property that the software will perform in a predictable way in response to conditions that are outside the range of behaviors expressed by the requirements, e.g., response to

"impossible" inputs from an external system. In addition to predictable, the response should be contained, i.e., a single invalid input should not necessarily cause the processing of input to stop.

20. To show that verification coverage of the requirements is achieved, measurable coverage criteria must be defined. These coverage criteria should take account of the logical structure of each requirement as well as the range of possible values for data elements referenced by the requirements.

21. Structural coverage, as opposed to requirements coverage, is a measure of the extent to which the executable object code is "exercised" by verification. Among other purposes, structural coverage may be used to identify unintended function (a.k.a. undocumented code). It may also reveal missing requirements, missing test cases, dead code and other anomalies.

22. Modified Condition Decision Coverage (MCDC) should be achieved unless it can be shown that there are one or more independent layers of protection (i.e., independently of the software) that reduce the risk associated with failure or unintended behaviour of this software. With just one independent layer of protection, decision coverage is sufficient. With two or more independent layers of protection, statement coverage is sufficient.

23. If a formal method is used to obtain verification results for the purpose of certification credit, it must be shown that the procedure used to obtain these results is correct in the sense that it strictly complies with whatever conditions may exist to ensure the soundness of results produced by the formal method. It must also be shown that the results do not depend on any assumptions that have not been validated, or supporting lemmas that have not been formally proved. As well, the verification results must provide sufficient detail so that the result could be independently produced, e.g., "Theorem Proved" is not sufficiently detailed.

24. In a conventional approach that uses testing, each verification result usually addresses only a few requirements. This makes it relatively easy to review the verification results to confirm that the results are correct with respect to the requirements. When a formal method is used for verifying source code or the executable object code for compliance with the requirements, the verification results must not be more difficult to independently review. In particular, a verification result obtained by formal methods may be rejected as unsuitable for the purpose of certification credit if it is intended to verify more than a few requirements such that the relationship between the verification results and the requirement is not easily discerned. Also, a verification result obtained by means of a formal methods may be rejected as unsuitable if the level of the verification result expressed is "too abstract" relative to the level of abstraction used to express the requirements.

25. In the case of verification results obtained by means of a formal method, it must be shown that verification results truly mean what they are intended to mean. For example, it must be shown that the verification results are not vacuous because of a "glitch" in the formulation of a theorem, i.e., the "false implies everything" problem.