A software product certification model for dependable systems
Heck, P.M.

Published: 01/01/2006

Document Version
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the author’s version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

Citation for published version (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 05. Dec. 2018
A Software Product Certification Model for Dependable Systems

Petra M. Heck

LaQuSo, Laboratory for Quality Software, an activity of Technische Universiteit Eindhoven and Radboud Universiteit Nijmegen
Den Dolech 2, P.O. Box 513
5600 MB Eindhoven, The Netherlands
+31 (0)40 247 2526
p.m.heck@laquso.com

ABSTRACT
An intuitive definition of a dependable system is a system that does not fail in unexpected or catastrophic ways. A software product without faults is the best guarantee for dependability. Our model indicates the checks that need to be done to retrieve as many faults as possible.

We believe that provable quality of dependable systems is based on the application of justified analysis techniques, if possible on formal methods.

The model can not only be used to certify products after their creation, but it also indicates which elements and relations should be present in a product that is being created. The model is easily converted into a checklist for both software developers and auditors.

Keywords
Dependability, Software Certification, Certification Levels, Formal Methods

1. INTRODUCTION
More and more applications depend on the reliability, availability, and integrity of ICT systems. Due to the increase of complexity at the hardware, software, and communication level, creating dependable systems has become a major scientific and engineering challenge.

Dependability is a system aspect that occurs in the field of embedded systems, communication systems as well as in the field of enterprise information systems. In all cases the ICT system is supporting or controlling another system that may fail due to physical causes, which has to be counteracted by the ICT system.

Next to proper methods for creation of dependable systems, the verification of such systems is also important. Many times the failure of dependable systems endangers human safety, so failure must be avoided at all cost. The developer of the system must verify the system before delivering it, e.g. through testing or manual review.

However, if systems are safety-critical and are to be used in a broad environment, an independent third party (not the supplier or the acquirer of the system) should verify the dependability of the system. A third party can produce an objective and complete judgment. A third party can even hand out certificates if the assessors use a standard way of producing the judgment.

In this paper we present a certification model for assessing the dependability of software. We do not consider hardware and network aspects of dependable systems, because they require very different methods of verification. We also do not consider other quality aspects like usability and performance because they are not top priority for dependable software, and different techniques (like ergonomic design or performance optimization) are needed to achieve them.

We believe that provable quality of dependable systems is based on the application of justified analysis techniques, if possible on formal methods.

Section 2 explains how we define dependability and what the main idea behind the certification model is. Section 3 describes the main concepts of the model. Section 4 till 7 work out the main concepts in detail. Section 8 discusses some related work and Section 9 is the conclusion.

2. DEPENDABILITY
The dependability of a system [22] is interpreted to mean its reliability, safety and security. The reliability is a property concerned with the functional behavior of a system; the security is a property concerned with the protection of classified information; and the safety is concerned with the prevention of disastrous consequence of the failure of the system.

Dependability is closely related to software quality. Characteristics like reliability, security, and availability are included in the standard for Software Product Quality, ISO/IEC 9126 [15]. Dependability can in this sense be considered as an area within product quality.
The primary goal of developing a dependable system is to ensure all the required functions are defined accurately and completely as well as implemented correctly. If the software is deployed in a critical system (e.g., a railway crossing controller) the system must also satisfy the related safety and security requirements.

We take on the view of Meyer [20] that dependability (interpreted to mean the same thing for software as reliability) covers the following three external factors:

- **correctness**: the ability to perform according to the specification;
- **robustness**: the ability to prevent damage in case of erroneous use outside of the specification;
- **security**: the ability to prevent damage in case of hostile use outside of the specification.

An intuitive definition of a dependable system is a system that does not fail in unexpected or catastrophic ways. A software product without faults is the best guarantee for dependability. Our model indicates the checks that need to be done to retrieve as many faults as possible. By fault we mean ‘an incorrect step, process, or data definition in a computer program (or its intermediate artifacts)’ [7].

### 3. MODEL CONCEPTS

We do not only consider the final software product (the source code or working system), but also intermediate deliverables like user requirements and detailed design. Each major deliverable is a **Product Area** in the model (see Section 4). We have identified different artifacts (see Section 4 and Figure 2) within the Product Areas, so that dependability can be investigated on a more detailed level.

There are four **Generic Goals**, which hold for all Product Areas: areas must be complete, uniform, correct and consistent. The Generic Goals are accompanied by **Generic Properties** that indicate how the achievement of the goals should be measured (see Section 1). There are four different achievement levels for each Generic Goal.

The Generic Goals can be translated into **Specific Goals** per Product Area that indicate what complete, uniform, correct and consistent means for that Product Area. The Generic Properties can be translated into **Specific Properties** per Product Area that indicate what the required elements and checks are to achieve a certain level of the Generic Goals (see Section 7).

![Figure 1: Concepts of the Certification Model](image-url)
When the achievement level for each of the four Generic Goals has been established, the overall Certification Level of the product can be determined (see Section 6). The more formal elements are present in the product and the more formal checks have been performed without detection of faults, the more dependable the product is considered.

The concepts of the model are summarized in Figure 1. Our concepts are loosely based on CMMI [3].

4. SOFTWARE PRODUCT AREAS

For our division of the software product into Product Areas we have taken the main deliverables of the development phases (requirements, high-level design, low-level design, implementation and test). We have split the requirements into a context description and a user requirements part, to emphasize the importance of the analysis of the system environment.

The model consists of six Product Areas:

- The context description (CD) describes the environment and main processes of the system.
- The user requirements (UR) specify what functions the system has to fulfill.
- The high-level design (HD) (also called software requirements) is a translation of the user requirements into a more precise description in terms of system architects.
- The detailed design (DD) consists of several models of the system that describe how the system will be built.
- The implementation (IMP) contains the system and its documentation, built according to the design.
- The tests (TST) describe the tests of the different software components and the whole system.

Each area can be further divided into subparts, which we call elements in this paper. These elements can be separate artifacts, a chapter within a document, or different parts of a larger model. For instance, the user manual will be a separate artifact delivered with the system, the non-functional requirements will be a separate section of the user requirements document, and the stakeholders can be described as part of the business process description (e.g. in the same diagram).

Figure 2 shows the areas, their elements and their interrelations. We have put the areas in the traditional V-layout. A line between two Product Areas means that elements in one area depend on a previous area in the V. E.g. High-Level Design is derived from the User Requirements, the System Test tests all functionalities in the High-Level Design, and the Acceptance Test can refer to tests reported in the System Test to prevent duplication of test cases.

The sub-elements will be explained in more detail in Section 7.

Figure 2: Software Product Areas with their Elements
5. GENERIC GOALS

Generic goals (GG) are goals that have to be achieved for each Product Area. They are translated into Generic Properties (GP) that hold for each Product Area.

There are four Generic Goals for all Product Areas:

[GG1] **Complete.** All required elements in the Product Area should be present and as much formalized as possible.

[GG2] **Uniform.** The style of the elements in the Product Area should be standardized.

[GG3] **Correct.** Each element should contain no internal consistency or correctness faults.

[GG4] **Consistent.** All elements should be consistent with each other and with the other Product Areas.

For each of these Generic Goals different achievement levels can be established, which we have summarized in Table 1.

The completeness of a Product Area (GG1) can be basic (all required elements are present, level 1) or extra elements may have been added. These elements can be semi-formal (level 2) or formal (level 3), which refers to the fact that they are specified in a formal language. The more formal an element is, the easier it can be subject to formal verification (less transformation is needed). For examples of semi-formal and formal elements see SP1.2 and SP1.3 in Section 7.1 till 7.6.

The uniformity of a Product Area (GG2) can be only within the Product Area itself (level 1), with respect to a company standard (level 2), or with respect to an industry standard (level 3). By industry standard we mean a general accepted description technique that is not specific for the company like the use of UML diagrams for design documents. If known standards are used, translations to formal methods are likely to be available, which makes formal verification easier.

The correctness and consistency of the Product Area (GG3 and GG4) can both be established with different means that gradually increase confidence: manually (level 1), with tool support (level 2), or by formal verification (level 3). For each Product Area the Specific Properties (see Section 7) will indicate with respect to what correctness and consistency will be verified.

From the levels in Table 1 and the Generic Goals we derive the Generic Properties: one for each goal that simply indicates that the level should be as high as possible:

[GP1.1] As many formalized elements as possible. Score 0 if any required element is missing; score 1 if any semi-formal element is missing; score 2 if any formal element is missing; score 3 if all elements are present.

[GP2.1] As much standardization as possible. Score 0 if elements of the same type have different style (e.g. if all use-case descriptions have a different structure); score 1 if elements of the same type have the same style; score 2 if all elements also comply with the company standard; score 3 if all elements also comply with industry standards.

[GP3.1] Zero faults with the most thorough check possible on internal correctness. Score 0 if any fault has been detected; score 1 if manual review of elements detects no faults; score 2 if tool-supported and manual review of elements detects no faults; score 3 if review of elements and formal checks detect no faults.

[GP4.1] Zero faults with the most thorough check possible on external consistency. Score 0 if any fault has been detected; score 1 if manual review of elements detects no faults; score 2 if tool-supported and manual review of elements detects no faults; score 3 if review of elements and formal checks detect no faults.

The Specific Properties indicate for each Product Area what the required elements, applicable standards and possible checks are. We will provide examples of Specific Properties for all Product Areas in Section 7.

### Table 1: Generic Goal Achievement Levels

<table>
<thead>
<tr>
<th>GG1: Complete</th>
<th>Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Some required elements are missing</td>
</tr>
<tr>
<td>1</td>
<td>All required elements are present</td>
</tr>
<tr>
<td>2</td>
<td>Semi-formal elements have been added</td>
</tr>
<tr>
<td>3</td>
<td>Formal elements have been added</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GG2: Uniform</th>
<th>Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No standardization</td>
</tr>
<tr>
<td>1</td>
<td>Within the product</td>
</tr>
<tr>
<td>2</td>
<td>Style complies to a company standard</td>
</tr>
<tr>
<td>3</td>
<td>Style complies to an industry standard</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GG3: Correct (within elements)</th>
<th>Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Faults are detected</td>
</tr>
<tr>
<td>1</td>
<td>Manual review/testing has not detected any faults</td>
</tr>
<tr>
<td>2</td>
<td>Automated testing has not detected any faults</td>
</tr>
<tr>
<td>3</td>
<td>Formal verification has not detected any faults</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GG4: Consistent (between elements)</th>
<th>Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Faults are detected</td>
</tr>
<tr>
<td>1</td>
<td>Manual review/testing has not detected any faults</td>
</tr>
<tr>
<td>2</td>
<td>Automated testing has not detected any faults</td>
</tr>
<tr>
<td>3</td>
<td>Formal verification has not detected any faults</td>
</tr>
</tbody>
</table>
6. CERTIFICATION LEVELS
From the four levels that have been achieved for the Generic Goals an overall certification level can be calculated.

The model indicates a certification level per Product Area. The certification level of the entire product can be determined by taking the minimum over the areas, but a Product Area-based certification is more informative. We can e.g. decide to only certify the Implementation Product Area if our interest is in the dependability of the final product without taking into account the development deliverables or testing deliverables.

The certification levels are based on an intuitive notion of when products are more dependable. A basic level is achieved when at least all required elements are present. The next level is when the required elements are present and uniform. We place external consistency above internal correctness because for the former all relations between elements need to be correct, which is more complicated than just considering the elements on their own. Finally, the highest level of dependability is achieved when a product is complete and uniform, and correctness and consistency have been verified with the most rigorous method.

The model has five certification levels. For each certification level we have indicated the level that is needed for each of the Generic Goals (see also Table 1):

1. **Initial**
   \( \text{GG1} \geq 1 \) and \( \text{GG2} = 0 \) and \( \text{GG3} = 0 \) and \( \text{GG4} = 0 \)
   Each of the required elements is present in the product.

2. **Standardized**
   \( \text{GG1} \geq 1 \) and \( \text{GG2} \geq 1 \) and \( \text{GG3} = 0 \) and \( \text{GG4} = 0 \)
   All elements are uniform.

3. **Correct**
   \( \text{GG1} \geq 1 \) and \( \text{GG2} \geq 1 \) and \( \text{GG3} \geq 1 \) and \( \text{GG4} = 0 \)
   All elements are internally correct and internally consistent.

4. **Consistent**
   \( \text{GG1} \geq 1 \) and \( \text{GG2} \geq 1 \) and \( \text{GG3} \geq 1 \) and \( \text{GG4} \geq 1 \)
   All elements are consistent with each other and with other Product Areas.

5. **Verified**
   \( \text{GG1} \geq 1 \) and \( \text{GG2} \geq 1 \) and \( \text{GG3} = 3 \) and \( \text{GG4} = 3 \)
   All elements and relationships have been verified with mathematically-based methods wherever possible, or the most rigorous method otherwise.

Note that this 5-step scale could be further subdivided. For instance we could say that “Industry standardized” (\( \text{GG2} = 3 \)) is more mature than “Company standardized” (\( \text{GG2} = 2 \)), or we could distinguish different certification levels of correctness, depending on the verification method used. To reduce complexity we have restricted the scale to 5 levels.

7. SPECIFIC GOALS AND PROPERTIES
Specific Goals (SG) are goals that hold for one Product Area only. Each Product Area has a different set of Specific Goals (although they convey some similarity as they are based on the Generic Goals). The Specific Goals are all implemented by one or more Specific Properties (SP) for the Product Area.

In the following sections we suggest a necessarily incomplete list for each of the Product Areas. The Specific Goals are a direct translation of the four Generic Goals to the Product Area. The Specific Properties that accompany them (the required elements, standards and checks) are different for each Product Area.

The Specific Properties indicate the elements or checks that are required for a certain Generic Goal level. For instance, to achieve level 2 for GG3 (correctness), all checks in SP3.1 (manual) and SP3.2 (tool-supported) need to be performed and should not reveal any faults.

The set of Specific Properties has been collected from literature, standards [1][4][6]-[14][19] and our own experience.

The complete model also contains comments and advice that e.g. indicate which tools to use or which formal methods are suitable to verify the properties. These comments also explain what an atomic requirement is, what the characteristics of a testable requirement are, etc. The comments and advice need to be included to make the model more usable and self-contained.

7.1 CONTEXT DESCRIPTION

**[SG1]** Complete. The context description is as detailed and as formal as possible.

**[SG2]** Uniform. The style of the modeled processes complies with standards in process modeling.

**[SG3]** Correct. Each element in the context description is described in a correct way.

**[SG4]** Consistent. The relations between the elements in the context description are correct and consistent.

We will first briefly describe the different elements we consider to be part of a complete context description:

- **Scope.** The scope establishes the boundaries of the target system (the system for which the software needs to be designed, e.g. a video recorder or a business unit).

- **Environment.** The environment the software system will be a part of, will work for. Questions to be answered are: ‘What are the processes that play a role in the environment and have influence on the target system?’ and ‘What functions does the target system fulfill for the environment?’.

- **Target System.** The target system itself. At least the objectives of the target system should be formulated and the main processes, specifically the ones that interact with the software system, will have to be described. An organization chart can be provided to...
see how the human resources are grouped and how the lines of command are.

- Stakeholders. The stakeholders are the entities (persons or organizations) that have some concern in the system. The stakeholders can be part of the target system or they may belong to the environment of it.
- Actors. Actors are users and other systems that will interact with the software system.

From this set of elements and the Specific Goals we derive a number of Specific Properties.

[SP1.1] Required Elements
- Boundaries of the target system;
- Functions of the target system for the environment;
- Main processes in the target system;
- Stakeholders;
- Actors.

[SP1.2] Semi-formal Elements
- Graphical overview (context diagram or use-case diagram) of target system;
- Flowcharts of processes
- Organization chart (if human resources are involved).

[SP1.3] Formal Elements
- Process model of target system.

[SP2.1] Uniformity
- Elements and documents of the same type have the same style.

[SP2.2] Compliance with Company Standards
- All elements and documents comply with company standards.

[SP2.3] Compliance with Industry Standards
- UML diagrams for use-cases;
- UML diagrams for processes;
- Dataflow diagram for context diagram.

[SP3.1] Internal Correctness
- Each actor is able to interact with the system.
- The textual description matches the diagrams.
- If applicable maintenance, support and disposal processes have been included.
- The process description is detailed to the level of interaction with the software system.

[SP3.2] Automated Correctness Checks
- Processes are validated by simulation.

[SP3.3] Formally Verified Correctness
- Process models are correct workflows (no deadlocks or dead tasks).
- Process models are mutually consistent.

[SP4.1] External Consistency
- Each actor is mentioned in the process description and all actors mentioned are included.
- The processes of the target system are enough to fulfill its functions for the environment.

[SP4.2] Automated Consistency Checks
- Diagrams are drawn in a modeling tool that shows the relation between actors and processes.

[SP4.3] Formally Verified Consistency
- The process models of the target system and the models of its environment are mutually consistent.

7.2 USER REQUIREMENTS

[SG1] Complete. The requirements are as detailed and as formal as possible.
[SG2] Uniform. The style of the requirements description complies with standards in requirements engineering.
[SG3] Correct. Each element in the requirements is described in a correct way.
[SG4] Consistent. The relations between the elements in the requirements description and with the context description are correct and consistent.

We will first briefly describe the different elements we consider to be part of a complete user requirements specification:

- Functional requirements or Use-cases. Functional requirements describe the functionality of the system from the perspective of the user. This can be done in plain text or in the form of use-cases. A use-case is a named "piece of functionality" as seen from the perspective of an actor. For each use-case several use-case scenario's are given (sequences of actions, events or tasks), the permitted or desired ones as well as the forbidden ones.
- Objects. Many types of entities play a role in the environment processes but only those that have to be represented in the system are collected. Not the individual entities, but only the types or classes to which they belong are listed (so not "client Johnson", but only "client"). The object description can be quite informal in the form of a glossary, or more advanced in the form of a data dictionary or object model.
- Behavioral properties. General behavioral properties are e.g. properties that express that certain conditions may never occur or that certain conditions should always hold. Usually these properties have a temporal aspect and therefore it is possible to express them in temporal logic, although a translation in natural language is essential for most stakeholders.
- Non-functional requirements. These are also called quality requirements. It is a set of different types of quality measures.

From this set of elements and the Specific Goals we derive a number of Specific Properties.

[SP1.1] Required Elements
- Functional requirements;
- Non-functional requirements;
- Glossary.
[SP1.2] Semi-formal Elements
• Data dictionary or object model;
• Use-cases (with scenarios);
• Flowcharts of processes;
• Behavioral properties.

[SP1.3] Formal Elements
• Relational diagram of data/object model;
• Process model of use-case scenarios;
• Behavioral properties specification, e.g. temporal logics.

[SP2.1] Uniformity
• Elements and documents of the same type have the same style.

[SP2.2] Compliance with Company Standards
• All elements and documents comply with company standards.

[SP2.3] Compliance with Industry Standards
• ERD diagram for object/data model;
• UML diagrams for use-cases.

[SP3.1] Internal Correctness
• No two requirements or use-cases contradict each other.
• No requirement is ambiguous.
• Functional requirements specify what, not how (no technical solutions).
• Each requirement is testable.
• Each requirement is uniquely identified.
• Each requirement is atomic.
• The definitions in the glossary are non-cyclic.
• Use-case diagrams correspond to use-case text.

[SP3.2] Automated Correctness Checks
• Requirements are stored in a requirements management tool which uniquely identifies them.

[SP3.3] Formally Verified Correctness
• The use-case scenario models are correct workflows (no deadlocks or dead tasks).
• The use-case scenario models are mutually consistent.
• The data model diagram is in normal form.

[SP4.1] External Consistency
• Each ambiguous or unclear term from the requirements is contained in the glossary.
• The use-cases or functional requirements are a detailing of the environment description in the context description (no contradictions). Each step in a business process that involves the system has been included in the requirements. Each task that the system should fulfill for its environment has been included in the requirements. All actors of the context description have been included in the requirements.
• Each object is mentioned in the requirements and all objects mentioned in the requirements are contained in the object model.
• For each object the create-, read-, update- and delete operations are covered in the user requirements or not applicable.
• The requirements do not contradict the behavioral properties.
• The use-case or functional requirements do not render the non-functional requirements impossible.

[SP4.2] Automated Consistency Checks
• Requirements and glossary/objects are stored in a requirements management tool which shows the relations between requirements, scenarios, actors, and objects.

[SP4.3] Formally Verified Consistency
• The use-case scenario models are compliant with the behavioral properties.
• The use-case scenario models are compliant with the non-functional requirements.
• The requirements description (process model and behavioral properties) complies with the environment description (process model) from the context description.

7.3 HIGH-LEVEL DESIGN

[SG1] Complete. The high-level design is as detailed and as formal as possible.
[SG2] Uniform. The style of the high-level design complies with standards in systems design.
[SG3] Correct. Each element in the high-level design is described in a correct way.
[SG4] Consistent. The relations between the elements in the high-level design and with the user requirements are correct and consistent.

We will first briefly describe the different elements we consider to be part of a complete high-level design specification:

• Software requirements. The software requirements are a more formal description of the use-cases and other user requirements. This is a mixture of text and process models. Process models can be made for the processes in the target system and for each use-case. The software requirements describe all the functionalities that the system offers.
• Object model. The logical object model is a picture of the objects that play a role in the system and their relations. Sometimes such a picture has already been made in the user requirements phase.
• Component model. The component model is a high-level division of the system into components or modules. The model includes the interfaces that the system must have.
• User interface. A first sketch of the user interface (if applicable) of the system is included in the design.
• Traceability matrix. This matrix demonstrates the link between the user requirements and the high-level design.
From this set of elements and the Specific Goals we derive a number of Specific Properties.

**[SP1.1] Required Elements**
- Software requirements;
- Objects and their relations;
- User interface description;
- Component division.

**[SP1.2] Semi-formal Elements**
- Object model;
- High-level component model (structure chart);
- Flowcharts of processes;
- Traceability matrix.

**[SP1.3] Formal Elements**
- Relational diagram of object model;
- Process models of system.

**[SP2.1] Uniformity**
- Elements and documents of the same type have the same style.

**[SP2.2] Compliance with Company Standards**
- All elements and documents comply with company standards.

**[SP2.3] Compliance with Industry Standards**
- ERD diagram for object/data model;
- UML diagrams or state charts for processes.

**[SP3.1] Internal Correctness**
- No two requirements contradict each other.
- No requirement is ambiguous.
- Each software requirement is testable.
- Each software requirement is uniquely identified.
- Process models correspond to requirements text.

**[SP3.2] Automated Correctness Checks**
- Processes are validated by simulation.

**[SP3.3] Formally Verified Correctness**
- The process models are correct workflows (no deadlocks or dead tasks).
- The process models are mutually consistent.
- The object model diagram is in normal form.

**[SP4.1] External Consistency**
- The software requirements are a more formal description of the user requirements; each user requirement is traceable to one or more software requirements.
- Each component is necessary for implementation of the requirements and no component is missing.
- Each object is mentioned in the requirements and all objects mentioned in the requirements are contained in the object model.
- The life cycle of each object is covered by the system.
- The traceability matrix contains all requirements.
- The user interface offers access to all described functionality.

**[SP4.2] Automated Consistency Checks**
- Requirements are stored in a requirements management tool which shows the relations between user and software requirements.

**[SP4.3] Formally Verified Consistency**
- The process models comply with behavioral properties.
- The process models comply with non-functional requirements.
- The process models comply with the process model from the user requirements description.

### 7.4 DETAILED DESIGN

**[SG1] Complete.** The detailed design is as elaborate and as formal as possible.

**[SG2] Uniform.** The style of the detailed design complies with standards in systems design.

**[SG3] Correct.** Each element in the detailed design is described in a correct way.

**[SG4] Consistent.** The relations between the elements in the detailed design and with the high-level design are correct and consistent.

We will first briefly describe the different elements we consider to be part of a complete detailed design specification:

- **Component structure.** This is a hierarchical structure where components or modules are split into other components and so on. At the lowest level we have the objects from the object model. The place of object or data stores is also indicated in the structure. The structure indicates which components communicate.

- **Component behavior.** The behavior of the components and the interaction between them is described in processes. Data types of objects and operations on the objects are specified.

- **System structure.** The system structure describes the mapping of the component structure to the software and hardware components of the system. The way this is done depends on the development environment. The mapping results in ‘skeletons’ of the components to be made in the construction phase.

- **Algorithms.** Optionally algorithms for the operations can be worked out in detail (otherwise this will be done in the construction phase).

- **Hardware/Software environment.** The environment is the mapping between the system structure and the physical hardware and software in the environment of the system. This mapping describes the infrastructure the system will operate on.

- **Interface specification.** This specification describes the formats of the interfaces between the system and its environment.

- **Traceability matrix.** This matrix demonstrates the link between the elements from the high-level design and the detailed design.
From this set of elements and the Specific Goals we derive a number of Specific Properties.

[SP1.1] Required Elements
- Component structure;
- Component behavior description;
- System structure;
- Hardware/Software environment;
- Interface specification.

[SP1.2] Semi-formal Elements
- Structure chart of component structure;
- Structure chart of system structure;
- Flowcharts of processes;
- Algorithms in pseudo code;
- Interface definitions in pseudo code;
- Traceability matrix.

[SP1.3] Formal Elements
- Process model of component behavior;
- Algorithms in formal language.

[SP2.1] Uniformity
- Elements and documents of the same type have the same style.

[SP2.2] Compliance with Company Standards
- All elements and documents comply with company standards.

[SP2.3] Compliance with Industry Standards
- State charts or UML diagrams for processes;
- UML Class diagrams for component structure;
- UML Deployment diagrams for system structure and environment.

[SP3.1] Internal Correctness
- The algorithms comply with their specification.
- All components have unique names.

[SP3.2] Automated Correctness Checks
- Processes are validated by simulation.

[SP3.3] Formally Verified Correctness
- The process models are correct workflows (no deadlocks or dead tasks).
- The process models are mutually consistent.
- Algorithms are proven.

[SP4.1] External Consistency
- Each logical component is mapped to at least one system component.
- The interaction (indicated relations) in the component structure is consistent with the interaction in the component behavior.
- All component interfaces from the component structure have been specified.
- Each software requirement is traceable to one or more logical components.
- The traceability matrix contains all software requirements and each logical component.

[SP4.2] Automated Consistency Checks
- Requirements and design are stored in a development tool that shows the link between requirements and design elements;
- The design is made in a drawing tool where links have been created between logical components and hardware/software components.

[SP4.3] Formally Verified Consistency
- The process models comply with behavioral properties.
- The process models comply with non-functional requirements.
- The process models comply with the interface specification.
- The process models comply with the process model from the high-level design.

7.5 IMPLEMENTATION

[SG1] Complete. The system and documentation are as detailed and as formal as possible.

[SG2] Uniform. The style of the system and documentation complies with standards in software systems and documentation.

[SG3] Correct. The implementation and documentation are correct.

[SG4] Consistent. The relations between the elements in the implementation and the documentation and with the detailed design are correct and consistent.

We will first briefly describe the different elements we consider to be part of a complete high-level design specification:

- **Software System.** The software system is the final end product. It contains (compiled) source code and all other software components that are needed to make the software system operational.
- **User Manual.** The manual is a document that describes how users should install and operate the system.
- **Documentation.** Any other documentation can be included with the system, such as a technical specification or a database model.

From this set of elements and the Specific Goals we derive a number of Specific Properties.

[SP1.1] Required Elements
- Software system;
- User manual or online help.

[SP1.2] Semi-formal Elements
- Technical specification;
- Database model.

[SP1.3] Formal Elements
- Process models of the system.

[SP2.1] Uniformity
- Elements and documents of the same type have the same style.
[SP2.2] Compliance with Company Standards
• All elements and documents comply with company standards.

[SP2.3] Compliance with Industry Standards
Not applicable due to variety in programming environments.

[SP3.1] Internal Correctness
• The user manual uses consistent terminology.
• The user manual can be used for instruction (list of tasks) or for reference (list of commands);
• The user manual contains descriptions of error messages and problem resolution.
• The user manual contains an index or search function.
• Warnings or cautions in the user manual precede the applicable step.
• Text in documentation corresponds to figures or diagrams.
• Manual code review.

[SP3.2] Automated Correctness Checks
• Detection of “bad practices” with static analysis tools.
• Compliance with coding standards checked by analysis tools.

[SP3.3] Formally Verified Correctness
• Static code analysis for e.g. null pointer exceptions and index-out-of-bound.
• The process models are correct workflows (no deadlocks or dead tasks).
• The process models are mutually consistent.

[SP4.1] External Consistency
• The system contains all components as described in the detailed design.
• The (user) interface descriptions from the design are consistent with the system’s interfaces.
• The documentation is up-to-date.
• The user manual describes all (critical) system functions.
• Each operation described in the user manual is executable on the system.

[SP4.2] Automated Consistency Checks
• The documentation is automatically generated from the system.
• The system (code) was automatically generated from the detailed design.
• The process models of the system correspond with the system (simulation).

[SP4.3] Formally Verified Consistency
• Static code analysis with preconditions, postconditions and invariants.
• The system (code) was automatically generated from the formal process models of the detailed design.
• The process models of the system comply with the process models from the detailed design.

7.6 TESTS

[SG1] Complete. The test documents are as detailed and as formal as possible.
[SG2] Uniform. The style of the test documents complies with standards in software testing.
[SG3] Correct. Each element in the test documents is described in a correct way.
[SG4] Consistent. The relations between the elements in the test documents and with the user requirements, the high-level design, the detailed design and the implementation are correct and consistent.

We will first briefly describe the different elements we consider to be part of a complete test specification:
• Test Protocol. The protocol describes the tests to be executed. It consists of the test procedures (steps to be taken) and the test cases (data input and expected output for the steps).
  o The unit & integration test describes tests of the different software components. Components can either be tested stand-alone (unit) or together (integration). Tests are based on the detailed design specification.
  o The system test describes tests of the whole system on functionalities. The tests are based on the high-level design.
  o The acceptance test describes tests of the whole system on suitability for the users. The tests are based on the user requirements and performed in the actual production environment.
• Test Report. The report describes the actual test results. Any deviations from the expected results should be clearly indicated.
• Traceability Matrix. This matrix indicates the relation between the tests and the user requirements, software requirements and detailed design.

From this set of elements and the Specific Goals we derive a number of Specific Properties.

[SP1.1] Required Elements
• Unit & Integration test protocol;
• System test protocol;
• Acceptance test protocol;
• Test reports.

[SP1.2] Semi-formal Elements
• Test scripts for test automation;
• Traceability matrix.

[SP1.3] Formal Elements
• System process models or testable specification (can refer to process models from other Product Areas);
• Paths in the process model or wrapper to system under test.

[SP2.1] Uniformity
• Elements and documents of the same type have the same style.
[SP2.2] Compliance with Company Standards
- All elements and documents comply with company standards.

[SP2.3] Compliance with Industry Standards
Not applicable. There is no standardized style for test documents.

[SP3.1] Internal Correctness
- Each test has a unique identifier.
- Tests describe steps to be executed.
- Tests describe expected results.
- Tests describe needed input.
- Tests also use boundary values, stress levels and erroneous input.
- Reports record actual results.
- Reports describe actual output.

[SP3.2] Automated Correctness Checks
- Test scripts are created automatically e.g. with capture-playback tools.

[SP3.3] Formally Verified Correctness
- The process models are correct workflows (no deadlocks or dead tasks).
- The process models are mutually consistent.

[SP4.1] External Consistency
- The execution of the test protocols on the system produces the expected results.
- There is a test report for each test in the protocols.
- There is a test in the acceptance test protocol for each user requirement.
- There is a test in the system test protocol for each software requirement.
- There is a test in the unit & integration test protocol for each (combination of) components.
- The traceability matrix contains all requirements, design components and tests.

[SP4.2] Automated Consistency Checks
- Test scripts are executed with test automation tools.
- Test scripts and reports are stored in a test management tool that links them together.

[SP4.3] Formally Verified Consistency
- The system specification is automatically executed against the system under test with tools like TorX (model-based testing) or the paths in the process model are executed on the system as test cases.

8. RELATED WORK
We have mentioned that our certification model was inspired by CMMI. CMMI is however for process maturity, not for product maturity. We did not find any other models that describe product quality in the sense of analyzing the correctness of a product. However, there are some other related findings in the area of product quality.

The software product maturity model by John Nastro [21] has three core elements: product capability, product stability, and product maintainability. Two sub-elements, product repeatability and product compatibility, are not universal to every software product.

He provides example measures for each of the elements like tests failed, changes per week, number of patches.

The Nastro model differs from our model in the first and foremost place because it only measures properties of the end product and not e.g. the requirements or the design. It also seems more appropriate for the tracking of development progress (i.e. compare builds within one project) than for the objective measurement of the product quality. As Nastro states the importance or weight of the elements and even the elements themselves may vary from project to project.

The ISO/IEC standard 9126: “Software engineering — Product Quality” [15] describes a two-part model for software product quality: a) internal and external quality, and b) quality in use. The first part of the model specifies six characteristics (see Figure 3) for internal and external quality, which are further subdivided into subcharacteristics. These subcharacteristics are manifested externally when the software is used as a part of a computer system, and are a result of internal software attributes. The second part of the model specifies quality in use characteristics. Quality in use is the combined effect for the user of the six software product quality characteristics. The standard also provides metrics for each of the quality characteristics to measure the attributes. An explanation of how this quality model can be applied in software product evaluation is contained in ISO/IEC 14598-1 [16].

An evaluation according to ISO/IEC 9126 is mostly based on metrics where our model uses a more rigid scale by providing yes/no checks. This yes/no scale leaves more room for expert opinions, but also caters for less precise comparison between two products. As our model focuses on correctness and consistency, ISO/IEC 9126 does not address consistency between elements as a separate concern. Correctness is in ISO/IEC 9126 mostly determined through indirect measures (e.g. measure the number of defects found during a production period). We therefore believe that our model is more suitable to determine product quality (correctness and consistency) whereas the ISO/IEC model is more usable to specify and measure the desired product quality (all six characteristics). In the future we could extend our model with other characteristics from ISO/IEC 9126.

The certification levels in our model are similar to the review levels in an article by Connie Clayton [2]. Clayton has identified the different levels in which documents can be reviewed to standardize the review process:

Level 1: Document completeness: individual document;
Level 2: Compatibility with standards;
Level 3: First-level consistency check: internal;
Level 4: Second-level consistency check: requirements check, external, minimal CASE tool usage;
Level 5: Major review: review code logic, algorithms, full use of CASE tools.

Before a higher level of review can be attempted, the steps of all previous levels must be completed. The accompanying checklists are highly specialized on the American Department of Defense related standards (DOD-STD-2167A and MIL-STD-498), so not always usable in general. Furthermore, some questions are subjective (“is the document legible?”) or hard to check (“is there any irrelevant data?”). The lower level checklists contain many precise questions but the higher levels are less well-defined.

Jakobsen et al. [18] describe a five-level maturity model for software product evaluation, where they apply the concept of a maturity model to product quality evaluations. They assume that product quality increases when evaluation methods get more mature (from basic testing against requirements to continuously optimizing the evaluation process). Level 2 (testing against basic requirements and achieving satisfactory results) is carried out by checking product’s conformance to the ISO 12119 standard. Parts of the maturity model have been incorporated in the ISO/IEC 14598 standard. As said, this maturity model focuses on the evaluation process and thus fundamentally differs from ours. We could however also use ISO 12119 as an additional standard to collect Specific Properties from.

Software certification as performed by e.g. the FDA [5] does not prove correctness. If a product receives certification, it simply means that it has met all the requirements needed for certification. It does not mean that the product is fault free. Therefore, the manufacturer cannot use certification to avoid assuming its legal or moral obligations. We proposed a certification model that does focus on correctness.

In summary we can say that we did not encounter any models that address product quality in the same sense that we do: related to correctness and consistency. There are however many approaches to software certification, that mostly rely on formal verification or expert reviews to determine the product quality.

We believe that our approach adds value with its comprehensiveness (from requirements to tests), its focus on correctness and by establishing a standard to perform software certification that also includes expert reviews and formal verification if necessary. It uniformly establishes what to check and how to check it. These checks are not new, but there is no place yet were they all have been put together into one model.

9. CONCLUSION AND FUTURE WORK

In this paper we have described a Software Product Certification Model. We have given examples for each of the Product Areas, but this list is not complete and in the final model will be accompanied by many more comments, advices and explanations. This is what we are currently elaborating on. Most of our input comes from case studies in industry, with real-size products and documentation.

The model can not only be used to certify products after their creation, but it also indicates which elements and relations should be present in the product when it is being created. Thus the model can be used by both software developers and auditors. The Specific Properties are easily converted into a usable checklist, for convenient scoring.

We claim that for a thorough software product certification, formal verification is necessary, but comes at the end of the process. It should start with more simple checks: are all elements present, are their relations consistent, are standards complied to, etc. Our model is comprehensive and flexible enough to allow certification of software products in all life cycle stages, with the applicable rigor for the criticality of the software, up to formal verification for the most critical products.

We continue to extend our model and apply it in industry case studies to demonstrate the added value of it, so that our Software Product Certification Model (SPCM) becomes recognized as a product standard.
10. REFERENCES


