Fraunhofer Institut
Experimentelles
Software Engineering

Inspection of High Level Statecharts

Authors:
Christian Denger

Funded by the German BMBF
under grant VFG0004A ("QUASAR")

IESE-Report No. 030.03/E
Version 1.0
April, 2003

A publication by Fraunhofer IESE

Fraunhofer IESE is an institute of the Fraunhofer Gesellschaft.
The institute transfers innovative software development techniques, methods and tools into industrial practice, assists companies in building software competencies customized to their needs, and helps them to establish a competetive market position.

Fraunhofer IESE is directed by
Prof. Dr. Dieter Rombach
Sauerwiesen 6
D-67661 Kaiserslautern

Abstract

Since their invention by Fagan in 1976, inspections proved to be an essential quality assurance technique in software engineering. Traditionally, inspections were used to detect defects in code documents, and later in requirements documents. However, not much is known how to apply inspections to design document. Statecharts are an important technique to describe the dynamic behavior of a software system. Thus, it is essential to define techniques for detecting defects in statechart models. In this report, checklists and reading scenarios are presented to support an inspector during defect detection for statecharts.

Keywords: Software Inspection, Software Review, Statecharts, Perspective-based Reading, Statechart Inspections

Table of Contents

1	**Introduction**	**1**
2	**Brief introduction to Inspections**	**2**
2.1	The Inspection process	2
2.2	Reading Techniques	4
2.2.1	Checklist based reading	4
2.2.2	Scenario and Perspective based reading	5
3	**Inspections of large scaled documents**	**7**
4	**State of the art in Statechart-Inspections**	**10**
4.1	Defect Taxonomy for Statecharts	10
4.1.1	Correct	10
4.1.2	Complete	11
4.1.3	Consistent	11
4.1.4	Unambiguous	11
4.1.5	Testable	12
4.1.6	Traceable	12
4.1.7	Feasibility	13
4.1.8	Understandable	13
4.2	Statechart Inspection with Traceability based Reading	14
4.3	Statechart Inspections with Checklists	15
4.4	The QUASAR-Inspection Approach	16
5	**Checklist-based Inspections of Statecharts**	**19**
6	**Scenario-based Inspections of Statecharts**	**22**
6.1.1	Perspective: Tester of the High Level Statecharts (System Specification Document)	24
6.1.2	Perspective: Low Level Statechart Designer (System Specification Document)	26
6.1.3	Perspective: Maintainer of the statecharts (system specification document)	29
7	**Conclusion and Further Research**	**32**
References		**33**

1 Introduction

Inspections are an industrial strength quality assurance technique that is widely used in almost all industrial domains. The inspection approach was first published by Fagan [Fag76] and focused on the detection of defects in code documents. Over the last decades, the approach was tailored to other software engineering artifacts, e.g. requirements document, test cases / plans, and design documents [GG93, Lai00, SE93, TSF99, Bas97, BGL96].

With the rise of object orientation, the inspection approach has to cope with new challenges. Now, an inspector has to deal with concepts like inheritance, distributed information, and abstraction. Even though, the literature provides a lot of references to the use of inspections in code, design, and requirements document, not much is known about how to use inspections to detect defect in object oriented design documents. Only a few approaches can be found that give advice on how to perform inspections in such design documents [TSF99, Lai00]. Since these approaches only focus on defects in all UML diagrams, i.e. in class diagrams, sequence diagrams, etc. they are too general to inspect each of this model type in detail. Thus, it is necessary to define techniques how to inspect more special aspects of object oriented design documents, for example, use cases, class-diagrams, and statecharts. Statecharts are a widely used technique to describe the dynamic behavior of a software system. Thus, this report will focus on the inspection of this diagram type. A technique, how to inspect use cases is described in an additional report [Sch02]. Furthermore, [Sch02] provides a generic approach how checklists and scenarios for perspective based reading can be systematically developed, which is adopted in this report.

This work is part of the QUASAR-project. In this project, a special form of statecharts, high level statecharts, are the major part of the system specification document. The approach described in this report supports the defect detection in these high level statecharts. Nevertheless, the inspection approach can be used to inspect every kind of statecharts that are part of the high level design.

The reminder of this report is structured as follows. Chapter 2 gives a brief introduction to inspections in general and frequently used reading techniques to support inspections. Chapter 3 addresses the problems of inspections in the large. Chapter 4 defines quality criteria that must be matched by statecharts and gives a brief overview of the state of the art in statechart inspections. Also the QUASAR inspection approach for statecharts is introduced. A checklist to support the inspection of statecharts is presented in Chapter 5. Chapter 6 provides reading scenarios to support perspective based reading of statecharts and Chapter 7 summarizes this report and gives hints for further research questions.

2 Brief introduction to Inspections

An inspection is a static quality assurance technique that allows the identification and correction of defects early in the software development cycle. In this report, the term inspection is used in the sense of a structured review process.. Other static quality assurance techniques like walkthroughs, management or team meetings are not considered as an inspection in this report.

The definition of an inspection as a static analysis technique to detect defects in software life cycle products is very abstract and therefore, this definition needs to be enhanced. In addition to the definition, the following aspects characterize an inspection:

1. It follows a defined process
2. The participants of an inspection have defined roles
3. The inspectors are supported by reading techniques
4. The participants in an inspection are trained
5. The results of an inspections are documented

These characteristics are explained in more detail in the following sections.

2.1 The Inspection process

In the following figure, the inspection process and the involved roles are summarized. The inspection process consists of four basic steps which are essential for a good inspection result. One or more roles participate in each step and one of these roles is responsible for the correct performance of the related step. Finally, the figure shows that in each step certain documents are produced or serve as input document. In the following paragraphs each step and the related roles are explained in more detail.

In the planning step, the organizer of the inspection is responsible for planning the whole inspection process. This activity includes the scheduling of the different process steps, to provide the document under inspection and all the other important document, e.g. checklists, reading scenarios, to the inspectors, reserving rooms for the meeting.

Figure 1: The Inspection Process

In the detection step, each inspector searches for defects in the document under inspection. The inspectors are supported by reading techniques, which are described in more detail in Section 2.1.2. During the detection step, all the issues, i.e. errors, questions, improvement suggestions, raised by the various inspectors are logged in a defect report document.

In the collection step the issues raised during the detection are merged into a defect collection documents during a meeting. The moderator is responsible for leading the meeting into the right direction. The aim of the meeting is to decide whether an issue raised by an inspector is a defect or not. Therefore, the author of the document under inspection shall participate in the meeting to answer questions or to clarify vague aspects in the document. The moderator must assure that the issues are not discussed too long (the duration of the meeting shall not exceed 2 hours, otherwise a second meeting should be scheduled) and that the inspectors evaluate the product not the author of the product.

Finally the author is responsible for the correction of the defects that the inspection team agreed upon in the meeting.

In order to assure successful inspections it is essential that all the results of the inspection are documented. For example, defect logs and effort sheets give valuable input for the evaluation of the effectiveness and the efficiency of an inspection. A second very important prerequisite for a successful inspection is that at least 50 percent of the inspectors are trained in inspecting software documents. Performing inspections with inexperienced inspectors is a thread for good inspection results. Moreover, certain roles in an inspection should be

trained in specific skills. For example, the moderator needs special moderation and special social skills to efficiently lead the inspection meeting.

2.2 Reading Techniques

A reading technique represents a series of steps or procedures that guide an inspector in acquiring a deeper understanding of the document under inspection and to detect defects in this document [LK01].

A lot of different reading techniques are distinguished in the literature. Examples are Ad-Hoc reading, checklist based reading, scenario based reading, reading by stepwise abstraction and function point reading. A complete summary of all these reading techniques can be found in [LK01]. Laitenberger defines a classification scheme for these reading techniques which allows a detailed comparison of the different techniques [Lai00]. As the comparison of different techniques is not the focus of this report, these aspects of the reading techniques are not discussed in detail.

In the following sections, two reading techniques will be discussed in more detail, namely, checklist based reading and perspective based reading.

2.2.1 Checklist based reading

Checklist based reading is the most frequently used reading technique. Within this approach the inspectors are supported by checklists that contain questions each inspector has to answer during the defect detection phase. These questions focus on certain quality aspects of the document under inspection. The checklist approach tells an inspector what to check but it provides only poor guidance how to check whether a certain quality aspect is fulfilled.

According to Laitenberger [Lai00] checklists have three basic weaknesses. First, the checklist questions are often extremely general, e.g. "Are the requirements complete?" Such questions are not useful to support the inspection process, since the inspectors are not guided how to verify the addressed quality factor. Second, concrete guidance is missing on how to use the checklist that is, when to answer a certain question based on which information. Third, the checklist questions are often not up to date, i.e. they are based on defects detected in the past but recent defect classes are not included in the checklists. Thus, complete defect classes might be missed during the inspection. An additional problem is that checklist are often too long; that is they contain to many questions.

According to different sources, e.g. [Lai00], [GG93], a checklist shall adhere to the following criteria in order to minimize the above mentioned weaknesses:

- paraphrased as precise as possible
- not longer than one physical page
- structured so that the quality aspect the questions are focused on is clear to the inspectors
- kept up to date
- focused on questions that reveal major defects
- a checklist question that is answered with "no" points out a potential defect
- derived from guidelines, rules, quality aspects used in the project context in which the artifact under inspection was created.

It is important to understand that checklists found in the literature are just examples that need to be tailored to the context in which they shall be applied.

2.2.2 Scenario and Perspective based reading

Beside a checklist, other approaches can be used to support the inspector during the defect detection phase. One technique, which shows in several experiments significantly better results regarding the efficiency of inspectors, is the scenario based reading technique.

The basic idea of this technique is that an inspector is guided by a scenario that tells him or her what to look for during the inspection and how to perform the inspection. Furthermore, the scenario assures that an inspector actively works with the document under inspection and thus he or she gains a deeper understanding of the document. This profound understanding of the document is a prerequisite to find more subtle defects in the document under inspection. Finally, the attention of the inspectors is focused on the essential parts of the document under inspection and thus avoid a cognitive overhead of the inspectors [Lai00].

A reading scenario consists of three main parts:

1. Introduction
2. Instructions
3. Questions

In the *introduction*, the goal of the scenario is described and the quality aspects that are most important in the particular scenario are defined. In the *instruction*, an inspector gets concrete guidance how to work with the document under inspection in order to detect defects and to gain a profound understanding. Furthermore, the instructions focus the attention of the inspectors to the essential information in the document. For example, the instruction part of a scenario that supports defect detection in a requirements document can state that an inspector shall derive a high level statechart diagram from the use cases. Finally,

the *questions* focus on common defect sources in a particular document or entity under inspection and thus, help the inspector to detect defects related to these questions while working with the document and to decide whether or not the document under inspection fulfills certain quality criteria [LK01]

In the current literature different approaches exist following the ideas of scenario based reading. These are defect based reading which organizes the scenarios around different defect classes, usage based reading, focusing on particular usage profiles during the inspection and perspective based reading that organizes inspection around the needs of the stakeholders of the document under inspection. In the following paragraphs the ideas of perspective based reading are described in more detail, since this approach is used in Chapter 4 to define scenarios supporting a inspection of statecharts

In the perspective based reading approach (PBR) the scenarios are defined according to different stakeholders of the document under inspection. The approach is based on the fact, that different stakeholders have different perceptions of the quality of a particular document. For example, for a customer of the future system has other needs on the requirements documents (understandability, completeness) than a tester who is responsible for deriving test cases from the requirements (testability, correctness). The PBR approach assures, that all necessary views on the document are considered during the inspection and thus, a maximum of possible defects can be detected. Due to this characteristic of PBR one of the essential steps in this approach is the identification of all important stakeholders of the document under inspection. If the perspective of one stakeholder is not considered in the inspection, the inspectors might miss essential quality criteria during the defect detection phase and thus critical defects might remain undetected [LK01].

The reading scenarios are tailored to the particular perspectives. The instructions describe activities that are typically performed by the stakeholder the scenarios is written for. For example, from the perspective of a tester, the inspector has to develop test cases from the document under inspection.

3 Inspections of large scaled documents

In the literature a lot of experience reports and experiments show that inspections are an efficient and effective way to detect defects. However, with the rise of more and more complex systems and larger documentation of such systems, inspections must cope with the problem that it is difficult to inspect the whole document. The rise of object orientation increased this trend, since inspections have to cope with new aspects e.g. inheritance, highly distributed information, i.e. different views on certain system entities. Furthermore, in an object oriented design document, a logical system entity is described by means of different diagrams types and the other way round, each diagram type describes various logical entities. Thus, there is a many-to-many relationship between a logical entity and its documentation. This results in usually very large documents under inspection, which make it even more difficult to perform a single inspection of the whole document.

These considerations raise the question, how to partition an object oriented system into parts that can be inspected. Earlier approaches suggested that the documents shall be partitioned into smaller chunks and then an inspection is performed for each document chunk. These approaches have two major drawbacks. First, in an object oriented development crucial information is distributes over several different documents or document parts. Thus, an inspector would miss essential information while performing the inspection. Second, these approaches are not scalable, i.e. they are not applicable for large scaled software projects. This results from the fact, that these approaches suggest size information as the main splitting criterion which causes severe problems as the different parts of such a partitioning approach are no longer self contained and conceptually incomplete. [Lai00, BL02, LA99]

Thus, another partitioning approach is needed for inspections of large scaled systems that assures that the parts of the document under inspection are self-contained and that the author won't miss necessary information during the inspection. Travassos et. al [TSF99] describe two approaches. The first approach suggests to organize the inspection around a subset of the most relevant system functionalities. The second one suggests to organize the inspection around conceptual entities in the system design, e.g. concepts appearing in the class diagrams, the interaction diagrams or the state machines. A third approach is presented by Laitenberger [Lai00, BL02, LA99]. He suggests an architecture-centric inspection (ACI) approach. The essence of this approach is, that the system is partitioned into smaller parts by use of the system's architecture. The architecture of the system is developed early in the development process. Even in the requirements phase the crucial logical entities of the system are known and

thus, the inspection can be organized around these entities. A second important aspect of ACI is, that in this approach the author distinguishes explicitly between a logical artifact of a software system and its documentation. This is a basic difference between ACI and earlier inspection approaches, as earlier inspection approaches are documentation centered as described above. However, also this approach does not state how to deal with large documents when the inspection team has to cope with time restrictions. ACI gives guidance how to separate the documents into smaller chunks but postulates to inspect the whole document.

[The02] presents an approach that is similar to the approach of Travassos et al. as it is organized around critical use cases of the system, called usage-based reading. During defect detection, the inspectors are guided by use cases and have to decide whether the inspected artifact fulfills the use case or not. A second important element in this approach is the assumption that defects are of different importance. Thus, the idea is that the inspectors focus on use cases most important to the users, since these use cases reveal the most "critical to user" defects. In order to deal with the problem of time restrictions, the use cases need to be prioritized according to their criticality. By doing so, it is assured that after the inspection those parts of the system are inspected that are most relevant for the user even in the case of time restrictions.

Even though, usage-based reading gives valuable hints on how to focus an inspection, it gives not much hints, how an inspector shall read the document and what an inspector shall look for. Therefore, a combination of the usage-based reading technique and the perspective based reading technique is suggested in this report. The ideas of usage based reading is to organize the inspection around prioritized use cases, addressing the problems of large document in combination with time restrictions. These ideas are combined with the ideas of checklist based reading and perspective based reading which give concrete guidance how to inspect an artifact and what to look for. Figure 2 clarifies this approach

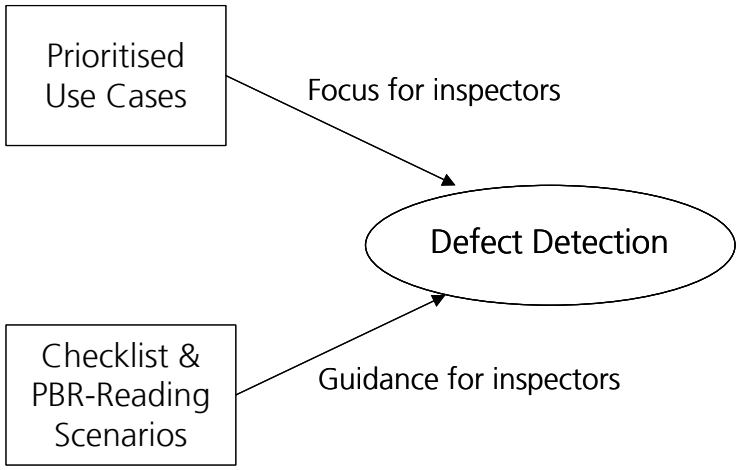

Figure 2: The QUASAR-Inspection Approach

In order to prioritize the use cases, two approaches can be utilized. First, the user and the customer can prioritize the use cases according to the relevance of the use case for the fulfillment of their most important goals. Such a prioritization can be done by means of interviews with the customer and the users of the system. The requirements engineering process requires a prioritization of the use cases anyway, then, the information is already at hand. A second opportunity is performing a risk analysis of the use cases in order to prioritize the use cases according to the evaluated risk level. In the first case, the use cases are ranked in decreasing order of the importance for the user and the customer, in the second case in decreasing order of the evaluated risks.

However, in the case that the document under inspection is small enough to perform an inspection of the whole document within time schedule, a prioritization is not needed.

4 State of the art in Statechart-Inspections

This section describes the results of a literature survey on inspection techniques for statecharts. There are only a few approaches in the current literature describing approaches for the inspections of object oriented design documents [Lai00, TSF99, Win97, MM99, Bin99]. Binder and Travassos et. al describe explicitly how to inspect statecharts. These approaches are described in more detail with respect to a defect taxonomy for statechart models.

4.1 Defect Taxonomy for Statecharts

In this section, a defect taxonomy is presented that is based on the IEEE standard 830 [IEEE] that describes quality criteria that shall be fulfilled by a good software requirements specification (SRS). The quality criteria are re-defined with respect to statechart models. Then, possible defects of statechart models are derived from the definitions.

Side Remark: In the defect taxonomy the quality aspect changeability, which is part of IEEE 830, is omitted. The reason to do so is that in several instances it is hard to distinguish for a defect between changeability and other properties in the table. For example, if something is hard to understand, should it be a defect related to Understandability or to Changeability. Therefore, in order to achieve orthogonal attribute values we omitted changeability.

4.1.1 Correct

Definition:
A statechart diagram or a set of diagrams is correct, if it is judged to be equivalent to some reference standard that is assumed to be an infallible source of truth.

Defects:
A statechart diagram or a set of diagrams is not equivalent to the Use Cases in the system requirements document; that is the statechart model is not consistent to the user requirements. Thus, the statechart diagram contains incorrect states, incorrect actions, incorrect events, incorrect conditions, incorrect transitions, and incorrect interactions of statecharts.

Example:
A use case of the user requirements is realized in the statecharts in another way

than needed; for example, the states of the statechart do not represent the states mentioned in the use case.

4.1.2 Complete

Definition:
A statechart diagram or a set of diagrams is complete, if no required elements are missing.

Defects:
A statechart diagram or a set of diagrams dose not contain all necessary elements. Crucial information that is required for subsequent activities is not presented. Thus, the statechart model does not represent all the user requirements; essential states, events, actions, operations, and guard conditions are missing; parts of guard conditions are missing.

Examples:
Missing states, missing operations/actions, missing events, missing transitions

4.1.3 Consistent

Definition:
A statechart diagram or a set of diagrams is consistent, if there are no contradictions among its elements or elements of other diagrams.

Defects:
The information of a single statechart diagram or the information of different statecharts is described in at least two different, incompatible ways so that there is a contradiction between different statechart diagrams, different statechart elements or between statecharts and other design diagrams (e.g., class-, sequence, collaboration- diagrams).

Examples:
The actions triggered by the same event, in two different statecharts, result in contradictory system behavior; guard conditions are not mutually exclusive; an event can lead into more than one destination state (not deterministic).

4.1.4 Unambiguous

Definition:
A statechart diagram or a set of diagrams is unambiguous, if every element therein has only one interpretation.

Defects:
The statecharts are ambiguous if elements of the statecharts can be interpreted in two or more ways. Thus, it is not clear, which of the two or more interpretations are true.

Examples:
The state-names are ambiguous, the event names are ambiguous (e.g. the event name "environmental input". In the case of more than one input form the environment, it is impossible to decide which event is meant in a certain situation), the action/operation names are ambiguous.

4.1.5 Testable

Definition:
A statechart diagram or a set of diagrams is testable, if there exists a feasible process to check that the statecharts fulfill their requirements.

Defects:
A statechart diagram or a set of diagrams is untestable, if there exists no feasible process to check that the statecharts fulfill their requirements. That means it is not possible to derive test cases from the statecharts due to logical or physical constraints.

Examples:
The expected system reaction in response to a certain event cannot be derived from the statechart model.

4.1.6 Traceable

Definition:
A statechart model is traceable, if it is possible in a syntactical sense to establish explicit links between each statechart element and the user requirements from which the statechart elements are derived.

Defects:
It is not possible to establish traceability links between statechart elements (state diagrams, states, events, conditions, operations, actions) and the corresponding use cases in the user requirements; that is the statechart elements are specified in a way that prohibits establishing explicit links.

Examples:
The statechart diagrams do not have unique identifiers. Then, it is not possible to link the statechart diagram to a certain user requirement.

Two essential quality aspects are missing in the taxonomy described above, namely, feasibility and understandability. The statecharts need to be feasible in the further development steps; that is, a designer should be enabled to easily transform, for example, high level design statecharts into low level design statecharts. Therefore, we add two additional quality aspects to those recommended in the IEEE standard 830 [IEEE]:

4.1.7 Feasibility

Definition:
A statechart model is feasible, if it is possible to transform the statechart model into lower level design models or code.

Defects:
The behavior described in the statecharts cannot be implemented; that means it is impossible to refine certain statechart diagrams into statecharts of a lower level of abstraction. The statechart diagrams and the statechart elements have too many relationships and communication links between each other. The statechart diagrams are too complex.

Examples:
It is not possible to derive low level statecharts from the high level statecharts.

4.1.8 Understandable

Definition:
The presented information is difficult to understand and comprehend. Specific instances are also deviations from the prescribed document format.

Defects:
The statecharts and statechart diagrams are difficult to understand and comprehend. The statecharts are not specified according to a given template.

Examples:
The statecharts are described too complex due to many relationships between the statecharts.

Note that the different quality criteria influence each other. For example, having a complete traceability between the use cases and the statecharts eases to realize changes in the statecharts or having a complete, unambiguous, and consistent statechart models positively influences the testability of the statechart models.

The mapping of the quality aspects to possible defects in the statechart models shows that the defects are related to states, transitions (including events, condition, and actions), operations and relations to other UML design-diagrams. The current state of the art, regarding statechart inspections, is presented in the following sections. Each approach is related to the possible defects in statechart models.

4.2 Statechart Inspection with Traceability based Reading

Travassos et al. [TSF99] describe a traceability-based approach (TBR) to inspect high level object oriented design diagrams using the unified modeling language (UML). More detailed information regarding the scenarios is described in an additional technical report [Shu99]. The approach uses scenario-based reading to support the inspectors during defect detection and focuses on the following design diagrams:

- class diagrams,
- interaction (sequence/collaboration) diagrams,
- state machines, and
- packages.

Although the approach is named traceability based reading, it focuses on other quality criteria if we follow the definitions given in the last section. The requirements on a system are considered as an important input for this inspection approach. Since the requirements are the basic input for the development of the above mentioned diagrams, it is necessary to check the correct and complete realization of the requirements in the high level design. Furthermore, the internal consistency of the diagrams themselves is of great importance, i.e. the consistency between the different diagram types. Therefore, Travassos et al. define two types of traceability based reading: horizontal and vertical reading. In this context *horizontal* means that diagrams on the same level of abstraction are compared to verify their consistency, for example class diagrams and statecharts. *Vertical* reading describes the comparison of documents on different abstraction levels to verify the completeness and the correctness of the high-level design diagrams; for example requirements and class diagrams are compared. The authors analyzed for which diagrams it is necessary to compare these against each other. They identified seven important combinations of documents that shall be considered in an inspection:

1. Comparing requirements description against class descriptions
2. Comparing use cases against sequence diagrams
3. Comparing class diagrams against class descriptions
4. Comparing class diagrams against sequence diagrams
5. Comparing state machine diagrams against class diagrams
6. Comparing state machine diagrams against sequence diagrams

7. Comparing state machine diagrams against requirements description and use cases

Each of these reading scenarios gives concrete guidance to detect completeness, correctness, and consistency defects in the corresponding documents. The scenarios guide the inspector during the comparison of the two documents under inspection. For example when comparing a requirements document and the state machine, the inspector shall read the requirements and the use cases in order to identify possible system states and events that might cause transitions between states. The states and transitions identified in this step are then compared to the states and transitions in the state diagrams. Thus, the inspectors can identify omitted, redundant, and incorrect states and transitions. All the other reading scenarios follow the same principle of comparison. The complete scenarios are described in detail in [Shu99].

4.3 Statechart Inspections with Checklists

In the literature a lot of checklists can be found for design inspections [Joh, NASA, Lai00, Bry99] but they are not specifically designed for defect detection in statechart diagrams. Binder describes several rules how to specify statecharts that are created to allow state based testing. Moreover several modeling style guides give hints on how well-formed statecharts should look like [Bin, UML, Amb02]. Such guidelines can serve as valuable input for the definition of a checklist. As the rules defined in Binder address specific quality problems of statecharts they are most interesting for our approach.

Binder [Bin] defines several rule sets on how statecharts shall be specified in order to guarantee their completeness, correctness, and consistency, before using them to produce a test suite. Therefore, Binder focuses on these quality aspects from a testers perspective. Some of Binder's rules are highly specific; that is, the rules focus on quality criteria that are relevant for detailed test design, for example, rules regarding state invariants. The approach postulates these rule sets as checklists that shall be used to perform inspections of statechart models that are used for stated based testing. These rules can be easily transformed into checklist questions.

Binder [Bin] defines the following five "checklists":

1. Structure checklist
2. State name checklist
3. Guarded transition checklist
4. Flattened machine checklist
5. Robustness checklist

The *structure checklist* defines criteria that assure the correct construction of statecharts, e.g. that every state can be reached from the initial state. The *state name checklist* focuses on weak, ambiguous, and inappropriate state names. Such names are often symptoms for misunderstood and ambiguous requirements. The *guarded transition checklist* focuses on the guards in the state transition. This checklists assures that the guard conditions are logically and structurally correct; for example, that all guard conditions are mutually exclusive. The *flattened machine checklist* focuses on potential defects in the class hierarchy, i.e. on the consistency between super-class and sub-class behavior in the case of inheritance; for example, that no state of the super-class is eliminated in the sub-classes. Also this checklist addresses several aspects that are very specific and hard to understand from a perspective other than a tester's perspective; for example, aspects regarding the state invariant of super class states. Finally, the *robustness checklist* focuses on necessary characteristics of the statecharts to assure correct and safe behavior under failure modes.

To summarize these aspects with respect to the quality criteria and defects specified in Section 4.1, Binder focuses on testability aspects of the statecharts. Each checklist is designed in a way that assures that it is easily possible to generate a test suite from the statecharts. Thus, ambiguity, completeness, correctness, and consistency criteria are assured with respect to the needs of a tester.

As mentioned in Section 2.1.2 checklists must always be tailored to the context of the project in which they are used to support the inspection process. Thus, the above checklists can serve as initial ideas for checklists for statechart inspections. The complete checklists can be found in [Bin].

4.4 The QUASAR-Inspection Approach

The inspection approach of this report is tailored to the context of the QUASAR-project. However, the approach can be used in any other development environment if the context is similar to the QUASAR context. Thus, the most relevant context factors of the QUASAR approach are briefly described in the following paragraph.

The QUASAR project deals with challenges on requirements engineering and quality assurance in the development of embedded systems, especially in the automotive domain. Beside other research questions, the improvement of quality assurance techniques is analyzed in this context. One important result of the QUASAR project are guidelines, describing how to transform requirements, specified as use cases, into a class diagram and corresponding high level statecharts [DKK02]. This transformation provides a more formal view on the requirements. Thus, the use cases in the system requirements document support the user/customer perspective and the class diagram and the high level statecharts in the system specification document support the designer perspective.

The idea is, that the high level statecharts (the system specification document) are used to support sub-contracting as the more formal view on the requirements facilitates the understanding of the requirements for the developers of the subcontracting company. In the QUASAR project, the tool Rhapsody in J is used to model the class diagrams and the high level statecharts.

The inspection approach described in this report focuses on quality assurance techniques for the designer perspective on the requirements, i.e. the high level statecharts. In particular, an inspection approach tailored to the needs of defect detection in high level statechart is presented. The QUASAR context factors influence the inspection process in several ways:

Certain aspects that should be checked in a statechart inspection are not considered in the approach due to the context setting. The check of the traceability and the consistency between the statecharts and the requirements (correctness) is not explicitly considered in the approach. Following the QUASAR guidelines, how to transform use cases into a class diagram and high level statecharts, assures these quality aspects in a constructive way rather than checking them afterwards. One can say that the developers might not follow the guidelines and thus, an explicit inspection of these aspects gets necessary. This is a valid argument but the guidelines can be easily transformed into a checklist and therefore, the approach focuses on other quality aspects. Moreover, if no constructive guidance is given, the TBR approach described in Section 4.2 should be used to compare the statecharts and the requirements as well as the statecharts and other design diagrams in order to assure these quality criteria.

The consistency between the class diagram and the statecharts is not explicitly checked in our approach, as the most quality aspects regarding the consistency between the two diagram types are assured by the use of the tool Rhapsody in J. For example, when using the tool, it is impossible that an operation or an event used in the statecharts is not defined in the class diagram. Also, some of the quality problems addressed in Binder's checklists, described in Section 4.3, can be automatically checked by the tool Rhapsody in J. For example, the flattened machine checklist addresses problems such as missing states in a subclass statechart in the case of inheritance. Such quality aspects are automatically assured by the use of the case tool Rhapsody in J, as the tool assures that the sub-class' statechart contains the same states and events than the superclass' statechart. However, if no tool is used for statechart modeling or a tool that does not assure such aspects, it is necessary to address these potential defect sources in the inspection process.

Note that the approaches described in the literature can only serve as a starting point to support the inspection of statecharts. Adapting these approaches to the concrete project environment is always necessary.

The QUASAR-Inspection approach focuses on quality criteria that are not addressed by current scenario based approaches to inspect statecharts. These quality criteria are testability, changeability, and realizability in latter development steps. In order to verify these quality aspects, perspectives based reading scenarios are defined that focus on the detection of defects that are related to these quality criteria. The following matrix summarizes how the different approaches are related to the quality criteria and consequently which defect types are addressed by the approaches:

Table 1: Qualtiy criteria in Relation to the approaches

	Travassos et al	Binder	QUASAR-Checklist	QUASAR-Scenarios
Correctness	**X**	**(X)**	**X**	**X**
Completeness	**X**	**(X)**	**X**	**X + tool**
Consistency	**X**	**(X)**	**X**	**X + tool**
Ambiguity		**(X)**	**X**	**X**
Testability		**X**		**X**
Changeability				**X**
Traceability	**(X)**			**(X)**
Realizability				**X**

Note that a "X" means that this quality aspect is directly addressed by the approach and a "(X)" means that the quality aspect is indirectly addressed due to relationships between the quality criteria. The entry "tool" shows which quality aspects are assured by using the tool Rhapsody in J in the QUASAR context.

If it is not possible to perform an inspection with the reading due to extreme time restrictions, a checklist to inspect the statecharts can be used. This checklist focuses mainly on syntactic aspects and contains high level questions. The matrix above shows that the checklist cannot be used to replace the reading scenarios as it does not address all the quality criteria, but it can supplement the perspective-based inspection.

5 Checklist-based Inspections of Statecharts

In this section a checklist for inspecting high-level statecharts is developed. This checklist is tailored to the QUASAR context factors described in Section 4.4 and therefore needs to be tailored to the specific context factors when used in another project. As described in Section 2.2, a checklist consists of several questions guiding the inspectors during the defect detection in "what" to look for. Following the guidelines how to describe a checklist given in [Sch03] the checklist presented below represents a starting point for checklist based reading of statechart models.

The checklist is designed in a way that it supplements the reading scenarios defined in Chapter 6. Moreover, the checklist is developed in the QUASAR context that is, the guidelines to transform use cases into statecharts need to be considered in the checklist, as it is important to verify that the developers of the statechart really applied the constructive guidelines. Thus, the questions 1 – 6 of the checklist assure that the constructive guidelines were used in the creation of the statecharts. Syntactical checks need not to be considered in the checklist questions, as the tool Rhapsody in J is used as a modeling tool. This tool automatically performs several syntax checks; for example, "Each statecharts has an initial state", "A junction connector has exactly one outgoing transition", "Each event is defined in the corresponding class", etc. If no case tool is used during statechart modeling the checklist needs to be enhanced with additional questions. This holds also in the case that a case tool with different features is used. Then, the checklist needs to be adapted to the features of the tool. The checklist questions 7 – 9 are examples how to enhance the checklist to cover such consistency checks that are usually performed by a tool.

The checklist was derived from several checklists found in the current literature [Bin99] and from style guides how to design statechart models [Amb02, UML]. The checklists recommended by Binder served as the main input for this checklist. Extracts of Binder's "Structure checklist", "State name checklist", and "Guarded Transition checklist" were used to define general checklist questions. However, some aspects of Binder's checklists are specific for testing statecharts and are too long. Therefore, Binder's checklists are generalized to support an inspector focusing on defects not specific for test considerations and that can be performed in a single inspection. However, if a detailed checklist-based inspection of statecharts shall be performed, it is recommended to consider all of Binder's checklists, tailored to the project context.

Completeness	✓
1. Are all the use cases considered in the statecharts?	
2. Is each user input, i.e. each monitored variable, mentioned in the use cases considered in the statecharts?	
3. Is each system output, i.e. each controlled variable, mentioned in the use cases considered in the statecharts?	
4. Is each exception mentioned in the use cases, realized in the statecharts?	
5. Is each rule mentioned in the use cases, realized in the statecharts?	
6. Are the names for the statecharts and the statechart elements consistent to the elements in the requirements?	
Consistency	✓
7. Is each statechart consistent to the class diagram; that is, are the events, operations, variables, and actions used in the statecharts defined in the corresponding classes?	
8. Is the statechart of a sub-class consistent to the statechart of the corresponding super-class?	
9. Has each statechart diagram a defined default state?	
10. Is each state of each statechart reachable from the statecharts initial state?	
11. Has each state, except the final state, an outgoing transition?	
12. Are all of the statechart elements (states, events, actions, guards, operations) needed to realize the behavior?	
13. Does each outgoing transition of a composite state make sense for all sub-states?	
14. Are for each state all the events and guards of outgoing transitions mutual exclusive?	
15. Does it make sense that with each recursive transition the entry and exit actions are re-performed?	
Correctness	✓
16. Are all the exceptions, rules and events mentioned in the requirements implemented correctly in the statecharts?	
Ambiguity	✓
17. Are all the state-, event-, action/operation-names meaningful in the application context?	
18. Are the state-, event-, action/operation-names of the statecharts unique?	

In the case of an inspection of a large document the recommendations how to partition the document for the inspection described in Chapter 3 shall be used.

We want to emphasize again, the checklist should be used to supplement the perspective-based scenarios or to perform a high level, more abstract inspection, focusing on syntactic aspects. The checklist shows that not all quality criteria, defined in Section 4.1, are addressed by the checklist. This results from the fact that a checklist usually focuses on defects resulting from syntactic aspects and subtle defects are hard to address by a checklist. Regarding the quality criteria and the related defects taxonomy described in Section 4.1 the checklist addresses ambiguity, completeness, consistency, and correctness problems on a high level of abstraction.

To address more subtle defects an inspector needs a profound understanding of the document under inspection. As described in Section 2.2, such an understanding can only be gained by actively working with the document. The perspective based reading scenarios ensure that the inspectors actively work with the document and thus focus on more subtle defects. However, also syntactic defects should be detected in an inspection with the reading scenarios. Therefore, the reading scenarios described in the following section are designed in a way that also syntactic defects can be detected. In the case that the checklist is used in addition to the reading scenarios, the reading scenarios need to be adapted in that way that the questions that address syntactic aspects are skipped in the perspective based inspection.

6 Scenario-based Inspections of Statecharts

In the QUASAR approach, high level statecharts are part of the system specification document. In this chapter, the perspective based reading scenarios for high level statecharts are described in detail. Following the approach described in [Sch02], the first step to develop perspective based reading scenarios is the identification of the relevant document stakeholders. That is, all the stakeholders who work with the system specification document in general and with the high level statecharts in particular must be identified. Furthermore, the most relevant quality aspects of these stakeholders regarding the requirements need to be identified. To support the inspection of the high level statecharts, three main perspectives are identified, as depicted in Figure 3.

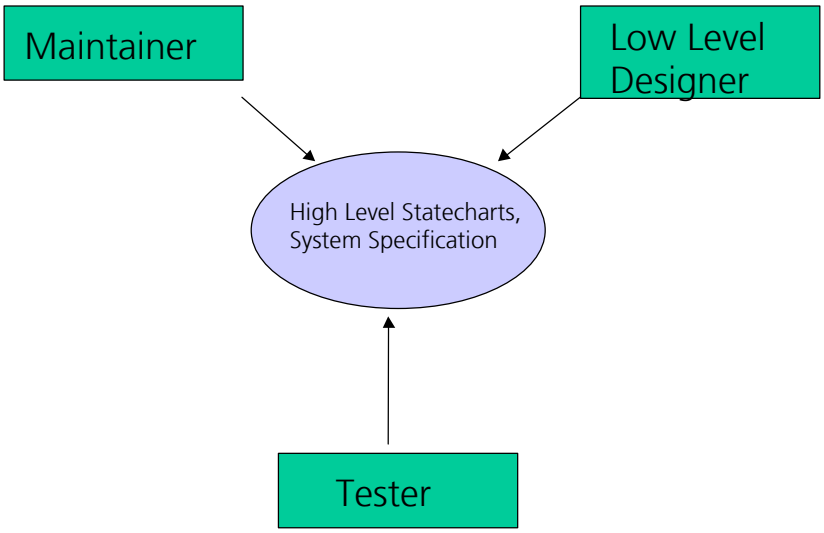

Figure 3: Stakeholders of the High Level Statecharts

The maintainer is responsible for realizing adaptive, corrective, and perfective changes in the high level statecharts. Therefore, the maintainer is most interested in easy to change statecharts; for example, that he or she can easily identify those statecharts and statechart elements affected by a change in the requirements.

The low level designer is responsible for refining the high level statecharts into the low level system design. By doing so, the designer needs to understand the high level statecharts and has to assure that all information needed to create the low level design is contained in the high level statecharts. Thus, this perspective is most interested in easy to understand, complete, and realizable high

level statecharts. In the context of the QUASAR project the requirements engineer assumes this role, as the low level statecharts are defined on the software requirements level and thus by the requirements engineer.

Finally, the tester is responsible for testing the high level statecharts. This activity contains the development of a test plan, the derivation of test cases from the statecharts, and performing the tests. Thus, the testability and the completeness of the high level statecharts are most important for the tester.

Having identified all the stakeholders and their needs, a particular reading scenario is defined for each stakeholders view. This supports the defect detection of an inspector assuming one of these views. Each reading scenario considers the goals, needs, and quality criteria that are characteristic for the corresponding view (stakeholder). The instruction step of each scenario, described in Section 2.2, is tailored to typical activities which the corresponding stakeholder performs with the document. In the following sections, each reading scenario is described in detail, following the structure for scenarios described in Section 2.2.

In order to derive concrete instructions for each scenario, some ideas of more general reading scenarios for object oriented design document [TSF99, Lai00, LA99] were used. However, the main part of the instructions is based on considerations how the different stakeholders work with the high level statecharts. These typical work-processes were transformed into instructions for the inspectors.

As described in [Sch02] each instruction part of a document is described by means of an agenda, i.e. each instruction has a unique number, a description of the activities to be performed in the step, and a validation column for each step. This validation column contains statements that assure that the related scenario step is performed correct and completely. Furthermore, the validation column contains questions regarding defects that can be detected in the corresponding scenario step. These questions ensure that defects are detected while performing the scenarios.

The reading scenarios described in the following sections are supported by the QUASAR context. Requirements management and in particular establishing traceability between use cases on the system requirements level and statecharts on the system specification level is addressed in the QUASAR project. In the QUASAR project, explicit links between the use cases and the classes and statecharts that realize the use cases are defined. Having these explicit links several steps of the reading scenarios can be supported:

- The complete scenario of the maintainer perspective
- Step 0 of each scenario that links use cases to statecharts

The traceability links facilitate the identification of related information elements in the statechart models and between the statecharts and other diagrams (class diagrams and use cases). Without the explicit links, the inspectors must search manually for the related elements. Thus, in the QUASAR project, the scenarios can be performed faster then in another context where traceabiltiy links are not available.

Note, that the scenarios described below can be used to inspect all high level statecharts of a document if this document is small enough. In the case of a large document, the inspector shall organize the inspection around prioritized use cases as described in Chapter 3. This approach can be easily used in the QUASAR context as use cases are transformed into statecharts and thus, the mapping between the prioritized use cases and the statecharts is easily possible. Furthermore, the explicit traceability between these documents established in the QUASAR context supports this selection process.

Note that the first step of each reading scenario can be skipped in the case that a document can be completely inspected.

6.1.1 Perspective: Tester of the High Level Statecharts (System Specification Document)

Introduction:
Assume you are a tester in a software project. As part of your job you derive test cases from the statecharts of the high level design description. Thus, from your perspective, the most important quality aspects are the testability of the statecharts and the completeness of the test cases.

Instructions:

<table>
<tr><th>Nr.</th><th>Step</th><th>Validation</th></tr>
<tr><td>0</td><td>Start with the use case with the highest priority. Specify the name of the use case in the first column of the following table. Then, identify all classes and high level statecharts that are related to this use case. Therefore, specify the name of the classes in the second and the names of the statechart in the third column of the following table.

<table>
<tr><td>use case name</td><td>class name</td><td>statechart name</td></tr>
<tr><td></td><td></td><td></td></tr>
</table>

Repeat the following steps 1 – 3 for as much use cases as possible. Perform these steps according to the priority order of the use cases, that is from high priority to low priority</td><td>All the classes and state-charts relevant to model the use case could be identified.

All states and classes are necessary to model the use case.</td></tr>
</table>

<table>
<tr>
<td>1</td>
<td>For each class in the system specification document identify the statecharts belonging to the classes. Thus, write the class name in the first column and the corresponding statechart names in the second column of the following table:

<table><tr><td>class name</td><td>statechart name</td><td>related classes</td></tr><tr><td></td><td></td><td></td></tr></table></td>
<td>All the statecharts belonging to the class could be identified.

All the statechart names and class names are unambiguous and meaningful in the application context.

All the states are needed to model the behavior of the class.</td>
</tr>
<tr>
<td>2</td>
<td>Identify all the classes in the class diagram that are related to the classes identified in Step 1 (Associations, Aggregation, Inheritance). Specify the names of these classes in the third column (“related classes”) of the table of step 1.</td>
<td>All associated classes could be identified.

All identified relation are reasonable.</td>
</tr>
<tr>
<td>3</td>
<td>Now, derive test cases for the statecharts identified in Step 1. follow these instructions. Make sure that each test case assures that the branches of at least all statecharts belonging to a specific class are covered.

Derive all the table entries by performing the test case manually.

1. Define the initial state of the system, before the test case is performed.
2. Define an event sequence that serves as an input for the system. Specify the name of each event in the first column of the following table.
3. For each event, specify the state of the system before the event is received in the second column of the table.
4. Specify the guards that must be considered for the event in the third column, if any.
5. With respect to the value of the guards, specify the action that is performed when the event is received, in the fourth column of the table.
6. Specify the expected state of the system after the event was received in the fifth column of the table.
7. In the sixth column, specify the expected state changes in the related classes you identified in step 2.</td>
<td>All the states, events, guards, and conditions to derive test cases can be found in the statechart diagrams.

Each test case assures that the branches of at least all statecharts belonging to a specific class are covered.

All the states in the statechart can be reached from the ideal state.

All the event-, action-, and guard names are unambiguous and meaningful in the application context.

All possible guard conditions are considered in the test case.</td>
</tr>
</table>

	Event	State	Guard	Action	Destina-tion State	State in assoc. class	
	Repeat the steps 1 – 7 as long as test cases for all statecharts identified in step 1 are specified.						

Questions:

1. Which information is missing to create the test cases?

2. Are the effects of the system reaction specified under all circumstances?

3. The evaluation of which guards might lead to side effects in the system?

4. In which cases is the behavior of the system non-deterministic; for example, due to guard-conditions that are not mutually exclusive?

5. Which events create in your opinion transitions into incorrect destination states? Please specify why.

6. In which transitions are the wrong actions performed or are actions performed that should not be performed at all?

7. In which cases are the transitions or the guard conditions un-reasonable from your point of view.

6.1.2 Perspective: Low Level Statechart Designer (System Specification Document)

Introduction:
Assume you develop the low level statecharts in a software project. To perform the transformation of the high level statecharts into low level statecharts one of your tasks is to get a profound understanding of the high level statecharts. Thus, the quality aspects understandability, completeness, and realizabiltiy of the high level statecharts are most important for your work.

Instructions:
Note that step 0 is optional (see Section 0)

<table>
<tr><th>Nr.</th><th>Step</th><th>Validation</th></tr>
<tr><td>0</td><td>Start with the use case with the highest priority. Specify the name of the use case in the first column of the following table. Then, identify all classes and high level statecharts that are related to this use case. Therefore, specify the name of the classes in the second and the names of the statechart in the third column of the following table.

<table><tr><td>use case name</td><td>class name</td><td>statechart name</td></tr><tr><td></td><td></td><td></td></tr></table>
Repeat the following steps 1 – 3 for as much use cases as possible. Perform these steps according to the priority order of the use cases that is from high priority to low priority.</td><td>All the classes and statecharts relevant to model the use case could be identified.

All states and classes are necessary to model the use case.</td></tr>
<tr><td>1</td><td>For each class in the system specification document identify the corresponding statecharts and sub-statecharts. Write the name of each class in the first column of the following table and the name of the corresponding statecharts and sub-statecharts in the second column.

<table><tr><td>Class Name</td><td>Statecharts und Sub-Statecharts</td></tr><tr><td></td><td></td></tr></table></td><td>All the statecharts and sub-statecharts belonging to the class could be identified.

All the statechart names and class names are unambiguous and meaningful in the application context.

All the states and sub-states are needed to model the behavior of the class.</td></tr>
<tr><td>2</td><td>In order to understand the interaction between different statecharts (and the corresponding classes) analyze the events of the statecharts. Therefore, write the names of the events of the statecharts into the first column of the following table. For each event, specify the name of the statechart in which the event is created. For each event, specify the names of all statecharts in which the event is consumed.

<table><tr><td>Event Name</td><td>Created in statechart</td><td>Consumed in statechart</td></tr><tr><td></td><td></td><td></td></tr></table></td><td>All events are identified at least in one statechart.

All the event names are unambiguous and meaningful in the application context.

All events that need to be consumed in a statechart can be identified and no superfluous events are identified.</td></tr>
</table>

<table>
<tr>
<td>3</td>
<td>In order to understand the behavior described in the statecharts create a table for each operation and action of each statecharts according to the following structure:

<table>
<tr><td>Operation / Action-name</td><td>Purpose</td><td>Defined in</td><td>Called in</td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
</table>

1. Write the name of each operation/action of the corresponding statechart into the first column.
2. Specify the perceived purpose of the operation / action in the second column.
3. For each operation/action of the corresponding statechart write the name of the class and the statechart in which the operation/action is defined into the third column
4. For each operation/action write the name of the class and the statechart in which the operation/action is called into the fourth column. Consider especially recursive transitions, that might trigger actions/operations several times.</td>
<td>All operations and actions of the statecharts are considered in the table, especially entry, exit and in-state actions.

The purpose of each operation/action could be understood when reading the statecharts.

All the operation/action names are unambiguous and meaningful in the application context.

Each operation/action that is called in the statecharts is defined in the corresponding class.</td>
</tr>
</table>

Questions:

1. Are there any interactions between statecharts for which you could not see the purpose? If so please specify why.

2. Can you imagine important interactions between the statecharts that were not considered? If so, please specify these interactions

3. Makes each outgoing transition of a composite state sense for all substates?

4. Which important information regarding events and operations/actions is missing that makes it impossible or difficult to refine the statecharts?

5. Are the calls of the operations/actions reasonable, especially recursively called entry, exit actions and operations on recursive transitions?

6. Which elements of the high level statecharts are difficult to realize in the low level statecharts? If any, please specify why.

6.1.3 Perspective: Maintainer of the statecharts (system specification document)

This scenario is slightly different form the previous scenarios, as the inspectors need particular input documents in addition to the document under inspection and the reading scenario. The inspector has to perform activities of a software maintainer, i.e. changes of the requirements have to be considered in the statecharts. In order to allow an inspector to perform this scenario, the inspection champion (i.e. the person who is responsible for inspections in the project) has to develop change requests for the requirements that are probable to occur during the development of the product. These virtual change requests should be specified for all use cases. If there are too much use cases in the requirements document, the prioritization of the use cases shall be used to identify those use cases for which a change request shall be defined.

Introduction:
Assume you are a member of the maintenance staff in a software project. One of your responsibilities is the management of requirements changes and to change the system specification consistently to these changes Therefore, the maintainability and changeability of the statecharts is one of the most important quality aspects from your perspective.

Instructions:
Note that step 0 is optional (see Section 0)

<table>
<tr><th>Nr.</th><th>Step</th><th>Validation</th></tr>
<tr><td>0</td><td>Start with the use case with the highest priority. Specify the name of the use case in the first column of the following table. Then, identify all classes and high level statecharts that are related to this use case. Therefore, specify the name of the classes in the second and the names of the statechart in the third column of the following table.

<table>
<tr><th>Use case name</th><th>Class name</th><th>Statechart name</th></tr>
<tr><td></td><td></td><td></td></tr>
</table>
Repeat the following steps 1 – 3 for as much use cases as possible. Perform these steps according to the priority order of the use cases, that is from high priority to low priority.</td><td>All the classes and statecharts relevant to model the use case could be identified.

All states and classes are necessary to model the use case.</td></tr>
</table>

<table>
<tr><td>1</td><td>Read the first change request in order to get a profound understanding of the change. Identify all the use cases that are affected by the change request. Write the name of each directly affected use case in the first column of the following table. Identify all the use cases that are related to the directly affected use case by analyzing the includes, extends, and interrupt relations of the use cases.

<table><tr><td>Use Case</td><td>Related Use Case</td></tr><tr><td></td><td></td></tr></table></td><td>All use cases that are affected by the change request could be identified.</td></tr>
<tr><td>2</td><td>In order to analyze which statecharts are affected by the change request identify for each directly affected use case and each related use case all the statecharts that participate in the realization of these use cases
Write the name of the statecharts into the second column of the following table:

<table><tr><td>Use Case</td><td>Related Use Case</td><td>Statechart name</td></tr><tr><td></td><td></td><td></td></tr></table></td><td>All the statecharts necessary to model each use case could be identified.

All the statecharts are needed to realize the use case.

All relationships between the use case could be identified.</td></tr>
<tr><td>3</td><td>For each statechart in the table of step 2, analyze the impact of the change request.

1. Specify the number of the change request in the first column of the following table.
2. Specify the name of the statechart that is affected by the change request in the second column.
3. For each statechart, specify all those elements (states, events, actions, guards, operations) that are affected by the change request in the third column.

<table><tr><td>Change Request. ID</td><td>Statechart Name</td><td>Affected Elements</td></tr><tr><td></td><td></td><td></td></tr></table></td><td>For each change request the affected statecharts and statechart elements could be identified.</td></tr>
</table>

Questions:

1. For which change request was it difficult (impossible) to identify the affected statecharts and statechart elements? Please specify why.

2. Are there change requests that are, in your opinion, difficult or impossible to realize? If so, please state why.

3. For which change request is it possible to reduce the affected statecharts and statechart elements? If any please state how this is possible.

4. Can you think of improvements of the structure of the statecharts that help to reduce their complexity? If any please state it.

7 Conclusion and Further Research

In this report, we presented an approach to detect defect in statechart models using inspections. A survey of the state of the art of statechart inspection showed that not much is published regarding this topic. Only two approaches deal explicitly with the topic of inspection of requirements statements. Thus, this report gives guidance how to apply the ideas of inspections to verify statechart models. In order to validate the approach, further research is needed. We plan to apply the checklist and the reading scenarios in a controlled experiment to evaluate their usefulness to detect defects and to continuously improve the checklist and the reading scenarios. In a second step, the approach needs to be evaluated in an industrial setting or in an industrial strength case study.

Another research question that should be addressed in future work is the question for which defect classes checklist based inspections are more suitable and for which defect classes perspective based inspections are more suitable. A first hypothesis is that checklists are more suitable to detect syntactical defects and reading scenarios are more suitable to detect subtle defects. This hypothesis should also be checked in a controlled experiment.

References

[Amb02] Ambler, Scott W.; Modeling Style Info, UML State Chart Diagramming Guidelines, Online tips and techniques for creating better software diagrams; www.modelingstyle.info; 2000

[Bas97] Basili, Victor R.; Evolving and Packaging Reading Techniques; Journal of Systems and Software 38 (1); 1997

[Bin99] Binder, Robert V.; Testing Object-Oriented Systems. Patterns, Models and Tools; Addison-Wesley Object Technologies Series; 1999.

[BGL96] Basili, Victor R.; Green S.; Laitenberger, Oliver; Lanubile, Filippo and others; The Empirical Investigation of Perspective-based Reading; Journal of Empirical Software Engineering, 2(1); 1996

[BL02] Bunse, Christian; Laitenberger, Oliver; Improving Component Quality through the Systematic Combination of Construction and Analysis Activities; In the Proceedings of Quality Week Europe; 2002.

[Bry99] Brykczynski, Bill; A Survey of Software Inspection Checklists; Software Engineering Notes vol. 24 No 1; ACM SIGSOFT; 1999.

[DKK02] Denger, Christian; Kerkow, Daniel; Knethen, Antje von; Paech, Barbara; Von Use Cases zu Statecharts in 7 Schritten; Technical Report at the Fraunhofer Institute for Experimental Software Engineering, IESE-Report No. 086.02/D; 2002.

[Fag76] Fagan; Michael E.; Design and Code Inspections to Reduce Errors in Program Development; IBM System Journal, 15 (3); 1976.

[GG93] Gilb, Thomas; Graham, Dorothy; Software Inspections; Addison-Wesley Publishing Company; 1993.

[IEEE] The Institute of Electrical and Electronics Engineering, Inc.; IEEE Recommended Practice for Software Requirements Specifications, IEEE Std. 830-1993; 1993.

[Joh] Johnson, Philip M; The WWW Formal Technical Review Archive; http://www2.ics.hawaii.edu/~johnson/FTR/

[Lai00] Laitenberger, Oliver; Cost-effective Detection of Software Defects through Perspective-based Inspections; PhD Thesis in Experimental Software Engineering; Fraunhofer IRB Verlag; 2000.

[LA99] Laitenberger, Oliver; Atkinson Colin; Generalizing Perspective-based Inspections to handle Object-Oriented Development Artifacts; In Proceedings of the 21th International Conference on Software Engineering; 1999.

[LK01] Laitenberger, Oliver; Kohler, Kirstin; Reading Techniques for Software Inspections; Technical Report at the Fraunhofer Institute for Experimental Software Engineering, IESE-Report No. 020.01/E; 2001.

[MM99] Major, Melissa L.; McGregor John D.; Using Guided Inspection to Validate UML Models; 1999 http://sel.gsfc.nasa.gov/website/sew/1999/topics/major_SEW99paper.pdf

[NASA] National Aeronautics and Space Administration; Software Formal Inspection Guidebook; Washington;1993. http://satc.gsfc.nasa.gov/fi/gdb/fi.pdf

[Sch02] Schlich Maud; Inspektion des Systemlastenheftes; Technical Report at the Fraunhofer Institute for Experimental Software Engineering; to be published (IESE); 2003

[Shu99] Shull, Forrest; Travassos, Guilherme H.; Carver, Jeffrey; Evolving a Set of Techniques for OO Inspections; Technical Report CS-TR-4070, UMIACS-TR-99-63; University of Maryland; 1999.

[SE93] Strauss, S.H.; Ebenau, R. G.; Software Inspection Process; McGraw Hill Systems Design&Implementation Series; 1993.

[The02] Thelin, Thomas; Empirical Evaluations of Usage-Based Reading and Fault Content Estimation for Software Inspections; PhD Thesis at the Department of Communication Systems, Lund University; 2002

[TSF99] Travassos, H. Guilherme; Shull, Forrest; Fredericks Michael; Basili, Victor; Detecting Defects in Object Oriented Designs: Using Reading Techniques to Improve Software Quality; In the Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA); Denver, Colorado; 1999

[UML] UML Semantics – Part; Behavioural Elements – Behavioural elements package: State Machines; http://etna.int-evry.fr/COURS/UML/semantics/semant11b.html

[Win97] Winter, Mario; Reviews in der objekt-orientierten Softwareentwicklung; Softwaretechnik-Trends 17:2; Mai 1997.

Document Information

Title:	Inspection of High Level Statecharts
Date:	April 24, 2003
Report:	IESE-030.03/E
Status:	Final
Distribution:	Public