Complete Requirements Through Analysis with Use Cases and State Charts

Ruediger Kaffenberger
FERCHAU Engineering GmbH
D-71522 Backnang, Germany
r.kaffenberger@ieee.org

André Zeh
F + S Informationstechnik GmbH
D-65549 Limburg, Germany
Andre.Zeh@flecsim.de

Abstract. We describe an approach to requirements analysis that is based on the combination of use cases with state charts that represent state machine models of the required system. The method has been developed to quickly gather a good set of requirements for car infotainment systems developed at Delphi Grundig, Nürnberg. We found that use cases and state charts essentially capture similar information. The tool support developed enables the generation of both views, use cases and state charts, from one set of data. In the method described the actors associated with use cases are replaced by their interfaces with the system. This not only simplifies communication of the requirements within the technical community but also supports the development of the state machine model and makes the method complementary to the approach for requirements completeness proposed in (Carson '98).

INTRODUCTION

A fictional but very realistic dialog between two mangers of a big car manufacturer. Manager 1: "Yesterday we informed all our dealers that we will not ship any replacement HAVAC[1] units for the new models to them any more." Manager 2: "Why that?" Manager 1: "Last quarter we had more than 8000 requests for spare units and of course quality assurance became alarmed and investigated the cases. They found that practically all replaced units had no defects and functioned according to specification." Manager 2: "I do not understand, 8000 of our customers have brought back their cars to the dealers for a replacement of the HAVAC unit but the units were not defective - how can that be?" Manager 1: "Curious isn't it? They claimed that the climate control would not switch to cooling, even after the car had been parked in the sun and was blistering hot. We found out that they were right; the system did not switch to cooling any more. It is a fault of the car owners, but it happens so easily that the designers in any case should have foreseen it: The drivers had the temperature control set for heating and subsequently pushed the button for maximum cooling. This locked the controller in a state where it would not turn on cooling. Only if you detach the control unit from the battery it will reset and function normally again." Manager 2: "What will you do about that?" Manager 1: "In the short run we will educate our dealers about what we found out, so that no more HAVAC units are replaced unnecessarily. And we are pressing our supplier to provide a software update. I hope we can put the cost of the call-back on them."

This fictional example illustrates the worst-case result of a development that was completed based on an incomplete set of requirements. It is well known that the cost of repair grows exponentially with the number project phases completed, but still in most projects the gaps in the requirements fabric are only searched and found in the late development phases, during system integration and test. Schedule overruns and excessive development cost are the results.

In the following sections we present a methodology that helps to systematically detect missing requirements.

[1] Heating, Ventilation, and Air Conditioning

A COMMON NOTATION FOR REQUIREMENTS

A common notation for requirements gathered from different sources. The starting point of our method development was the task to find a way to describe the requirements of a car radio in a common form.

Car radios today are complex, function rich systems. They are tightly integrated with many other sub-systems of the car and thus a large number of constraints and requirements exist. The information about these requirements is scattered over a large number of specifications, typically supplied by the car manufacturer. To get a clear picture of the required system all these requirements had to be assembled in a structured way and presented in a common form. The bulk of the requirements in question described parts of the radio's behavior and many of them involved a car passenger who interacts with the radio's controls. Thus the decision was to employ use cases as the common form of requirements notation.

State machine models to analyze the situation dependency. While a car radio can perform many different functions, not all of these functions are available at all times. Depending on the situation, only sub-sets of the functionality are available to the user. Moreover the radio displays different behavior to the same stimuli, depending on the history of activities and situations. Use cases alone would either not be sufficient to describe all this or would become hard to read and understand. The situation dependency had to be described through state-machine models and state-charts were chosen as the notation for the state-machine models.

THE USE CASE NOTATION

The use case form of requirement notation. The use case notation for requirements has become popular in the software community in the wake of the object oriented methods and has always been part of UML. Use cases did not find entry into the systems engineering community easily. Main reasons for this are the impression that they are applicable in an object-oriented context only and the fact that in the UML 1.3 specifications (OMG '99) only use case diagrams and a couple of relations are defined. There is no sufficient information about the elements that make up a use case and their notation. As with all popular notations, in the literature a large number of different approaches to the notation can be found. Because of use cases are not so common in systems engineering and because of the competing definitions in literature, below we give a short introduction to use cases, their elements and structure:

Use case diagram. The use case diagram - with the stick-figures and ellipses in a box - is the most well known element of the use case notation. Often there is the understanding that this diagram is the use case. But this is not true. The use case diagram is the context diagram for

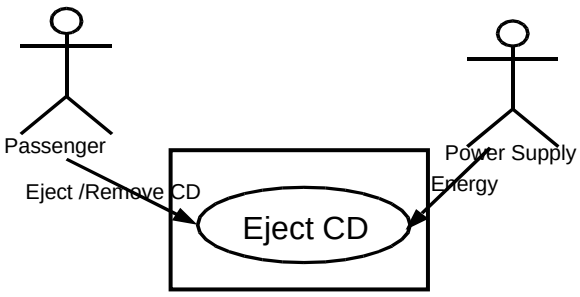

Figure 1, Use case diagram

one or more use cases. It shows actors - the stick figures, the system boundary - the box, and use cases - the ellipses. Arrows mark the relations between actors and use cases and between use cases and use cases.

Actors. An actor is a neighboring system that interacts with the system that performs the use cases. All stakeholders are actors at some time.

Use case. A use case represents the way the system employs to produce the value for the actor and all ways how it may fail to provide this value. A use case consists of a couple of elements:

Purpose or goal. A statement that describes the value or service the actor expects from the system.

Preconditions. A system may not always be able to potentially provide the expected service. The service may be available only under certain conditions and under all other conditions the service is not available.

Trigger. Usually a stimulus that starts the process that produces the desired service or value, especially if under one set of preconditions it potentially can provide more than one service.

Postconditions. After the system has provided the service or has in some way failed to provide the service the system's state may have changed and, for instance, it may not be able to provide the same service again without some intervening activity.

Results. In many cases a system delivers its service in well defined steps of completeness.

Use case steps, sequences, scenarios. The terms “Use case steps”, “sequence”, and “scenario” can be used interchangeably. They denote the processing steps the system performs and the interactions it has with its environment when it tries to provide the desired service. There may be different sequences that lead to the same result or sequences that lead to different results.

Replacing actors by interfaces. When writing or reading use case descriptions many associate actors with persons that interact with the system. This association is emphasized by the stick-figure used to symbolize the actor in the graphical notation of the use case diagram. This mental model works quite well if one discusses a business system. It does not work very well in the context of technical systems as they are typical for systems engineering tasks. It is simply hard to speak of a power supply or a lightning that strikes a wire as an actor. Another problem arises if there are different classes of actors. For instance if there are actors passenger, service technician, and test engineer, and they all give the system the same stimulus: Do we talk of different use cases or do all those actors belong to a common super-class of actor and we do talk of a single use case? These are questions that very often arise in reviews, especially if some of the participants are new to the method. The problem with the mental model of actor, that is inappropriate in the context of technical systems, can be avoided if actors are replaced by interfaces.

Actors can be replaced by interfaces because all stimuli have to enter or leave a technical system through an interface. Instead of thinking that an actor interacts with the system in this or that way we can think that an interface becomes active. The way how the interaction takes place is defined through the protocol associated with the interface. This much more technical view on the system to system interaction focuses the effort to the scope of the problem to solve: what requirements exist for the system to be built, and shields from the subtleties of the outside world. Of course this approach also incorporates some dangers as will be shown later.

Use case: Eject CD
Goal: There is no CD in the radio's CD-player.
Preconditions: The radio is in the sleep state or the radio is in the tuner mode or the radio is playing a CD (CD-mode).
Trigger: The interface Eject Button has been activated.
Postconditions:
For **result success**: The radio is in the sleep state and there is no CD in the CD-player or the radio is in the tuner mode and there is no CD in the CD-player.
For **result failed**: The radio is in the sleep state and there is a CD in the CD-player or the radio is in the tuner mode and there is a CD in the CD-player.
Scenario:
(Extension point 1)
The radio ejects the CD through the CD loading slot. The edge of the CD protrudes at least 20mm from the radio's face plate and at most 25mm.
The CD has been removed from the CD loading slot within 30s after the CD has been ejected.
Exception: The CD has not been removed from the CD loading slot within 30s after the CD has been ejected.
The CD is reloaded.
(Extension point 2)
Extension 1 (the radio is in the sleep-state): The radio is awoken from sleep state within 0.5s.
Extension 2 (the radio was in the sleep state): The radio is put to sleep state within 0.5s.

Figure 2: Use case notation

Example use case. The example figure 2 shows the use case notation for a requirement on a car radio to provide the functionality to eject the currently loaded CD. The example shows two additional elements of the scenario notation, extensions and exceptions. Exceptions describe deviations from the normal sequence due to exceptional conditions; in this case the CD has not been removed as expected. Extensions are used to describe additional steps necessary under certain conditions. The extension point marks the point in the sequence where the additional steps are to be inserted.

STATE CHART NOTATION

The state machine form of requirements notation. The state machine has a long tradition in electrical engineering and systems engineering. The most important notation is the state chart notation, developed by Drusinski and Harel (Harel '87). A subset of the state chart notation is included in UML (OMG '04).

Figure 3 shows a state chart. It specifies the radio's basic behavior when the following events occur: *Turn on*, *Turn off*, *CD button pressed*, *tuner button pressed*, *eject button pressed*, *CD inserted* and *CD removed from slot*. In the state chart notation states are represented by rectangles with rounded corners. A state may have sub-states. A very important concept is that of orthogonal sub-states. In the example the states *Eject mode* and *Normal mode* are orthogonal to the states *Safe* and *Hangout*. That is at the same time the system can be in one of the states *Eject mode* or *Normal mode* and in one of the states *Safe* or *Hangout*.

The arrows represent transitions between states. Transitions are labeled by the trigger that stimulates the system to perform the transition and possibly by additional conditions (guards) that also must be met for the transition to take place. The guard expression is written in parentheses. An example for a guarded trigger is the transition from *Tuner mode* to *CD mode*. This transition is only possible if the CD player contains a CD and thus the system

is in state *Occupied*.

In the example some additional pseudo-states occur: The initial pseudo-state (a small black ball with a transition at one end) indicates the transition to sub-states if the super-state is reached. The history-state (a circumscribed "H") means that the system remembers the last active sub-state. From the conditional pseudo-state (circumscribed "C") only one of the

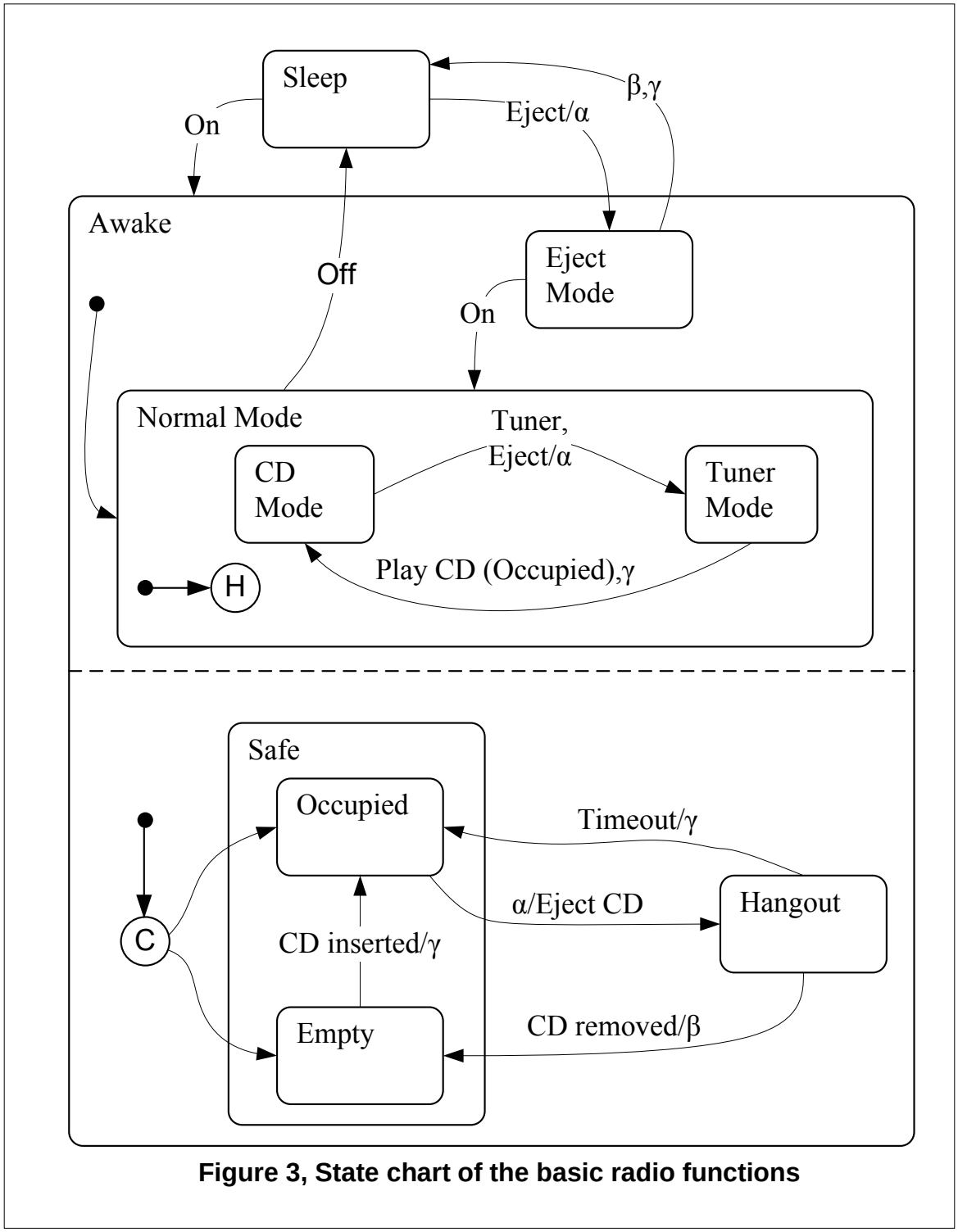

Figure 3, State chart of the basic radio functions

transitions are possible depending on guards.

Where do we find what the system does? Generally the notation allows to associate the production of some output (output is roughly equivalent to service) when the system enters a

state, leaves it, or while being in a state. If it is necessary the state rectangle is annotated with the output produced. Additionally the production of output can be associated with a transition, that is, the output is produced during the transition. In the example of figure 3 the transition from state *Occupied* to state *Hangout* produces the output that the CD is ejected. In the notation an output produced during a transition is noted on the transition arrows, labeled after a slash character that separates it from the trigger.

Output may not necessarily leave the system. The events α, β, and γ shown in figure 3 are events produced by the system for internal use only. These events synchronize the orthogonal components of the state machine model. For instance, during removing the CD from the slot (transition from state *Hangout* to state *Empty*) the internal event β is generated. If the system is simultaneously not in *Normal Mode* but in *Eject Mode*, this event will trigger the transition from state *Eject Mode* to state *Sleep* and consequently switch off the device.

THE GAIN FROM COMBINING USE CASES AND STATE CHARTS

Comparison between use case notation and state chart notation. The two examples figure 2 and figure 3 very much represent the same requirements. The two notations have a number of common elements but also a number of differences. The most obvious difference is that the state chart notation uses graphics while the use case notation is entirely text based, but it is obvious that a state chart could be represented by structured text as well. The more important difference is, that one use case represents only one aspect of the system behavior (the process of ejecting the CD in the example) while one state chart is able to represent all aspects of a system's behavior simultaneously, as has been implied by the *On/Off* and *Play CD/Tuner* transitions in figure 3. Obviously a number of additional use cases are necessary to describe the full system behavior covered by figure 3.

Most of the common elements of the notations are obvious: Both notations use states and triggers. The transitions between states are comparable with the use case's scenarios and the outputs of the state machine model that leave the system are equivalent to the transactions with the actors' respective interfaces during the use case's scenario.

Complementary notations. Use cases focus on the way a certain service is delivered and with what actors or through what interfaces the system interacts with its environment to deliver that service. State charts focus not on a single service but on the set of states that have to exist and the transitions that have to be performed for the system to be able to deliver the full set of required services.

Use cases are rather narrative and informal. This characteristic makes them suitable for the communication with people that have less technical or mathematical background. Because of their formal simplicity use cases are well suited for the capture of the first set of basic requirement on a system.

State charts in contrast are based on a mathematical model. Thus they are very formal. This makes state charts ideal for the verification and refinement of the information captured with use cases. Ambiguities and missing cases are easily spotted. In the example of figure 2 for instance the use case description missed the case that the CD player already had been empty and it did not define how the radio should behave when the eject button was pressed while it was playing the CD.

The mutual refinement process. The complementary nature of use cases and state charts leads to the mutual refinement process shown in figure 4. For example if the requirements analysis starts with some activities or services expected from the system, the respective use cases are constructed. The preconditions and postconditions found for the use cases then are examined in the state chart view. Missing transitions to and from states are added in this view and elaborated after a switch back to the use case view. This circular process is repeated until no new information can be added.

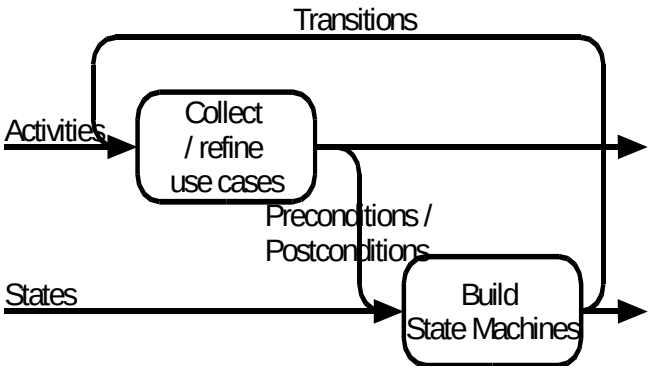

Figure 4, Mutual refinement of state charts and use cases

Very often the analysis can be started with a state based description of the system's behavior. When the transitions are viewed as use cases nearly always additional preconditions and postconditions are found, that make the state based description more complete.

By alternately analyzing the system in the use case view and in the state chart view with only a few iterations a very complete set of requirements is found. Not only the functional

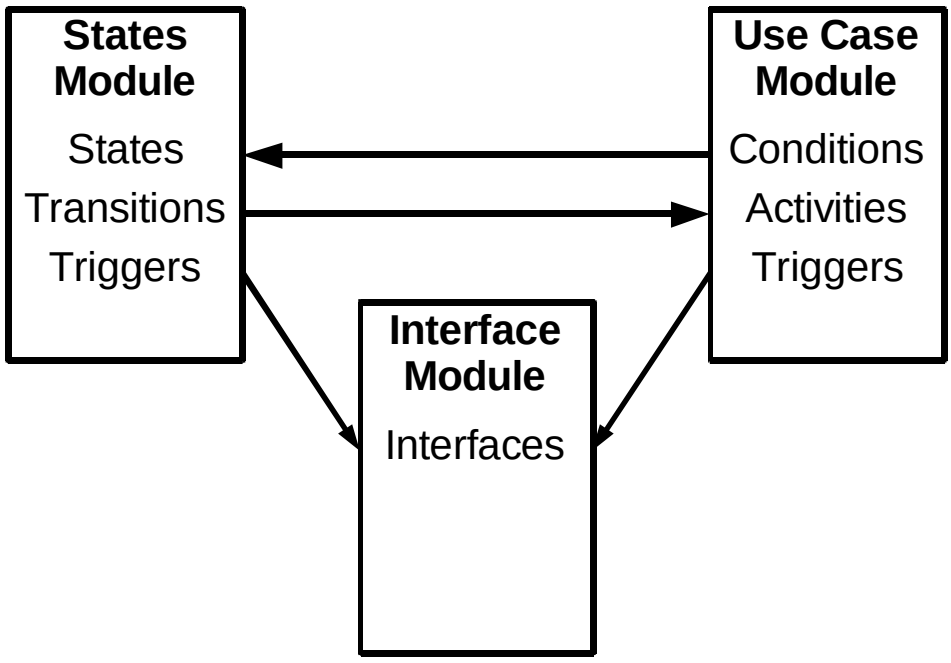

Figure 5, References between modules

requirements are found but also a wealth of information about the required performance, because, as has been hinted in the example of figure 2, the use case description makes it easy for see and attach critical performance parameters.

IMPLEMENTATION OF THE METHODOLOGY

Redundancy. While having two representations for the same information is a big advantage for the analysis of the information, from the standpoint of data management redundant information is a big problem. It would be a lot of work, if the information had to be entered into both models separately and keeping that information consistent could turn out to become a nightmare. The core of the requirements analysis method outlined above is the ability to see the identical information from two standpoints and to add missing information to the two views in one step. Any implementation must support this.

Tool support. Obviously an implementation of the method requires tool support. Requirements management tools can represent use cases (Kaffenberger 2001) and state charts, when put into a textual form, can be represented with a requirements management tool too. Thus we decided to employ DOORS for the implementation. This gave us the additional benefit that direct traceability could be established from customer requirements, to use cases and state charts, to system architecture and to test procedures. To capture the requirements in the form of use cases and state charts we use three modules (figure 5).

The state module mainly contains the definition of the state hierarchy. The transitions are represented by references to activities and the triggers by references to interfaces that become active to start a transition. The use case module mainly lists the activities associated with the use cases. Preconditions and postconditions are represented by references to state definitions in the state module. Interfaces are referenced as triggers and in the description of the scenarios. The interface module contains the descriptions of the interfaces.

Both the state module and the use case module provide a completely human readable and understandable view, even without reading the other module. The state module provides a state chart view. It combines the information from the state definitions with the referenced information from use cases and interfaces into a textual representation of a state chart (figure 6). The use case module provides a use case view. It combines the use case specific information, activities in the use case scenarios, and the referenced information about states and interfaces into the usual textual use case representation as shown in figure 2.

ID	System State Chart	Type	State Description
SSD_26	3.1. Sleep Radio consumes only minimum power.	State	3.1. Sleep Radio consumes only minimum power.
SSD_51	<trigger><activity><state>	Transition	Trigger: On Activity: Power-up radio Next: Awake {(Normal Mode)/(C Occupied,Empty)}
SSD_52	<trigger><interface><state>	Transition	Trigger: Eject Activity: active α Next: Eject Mode
SSD_27	3.2. Awake {(Normal Mode)/(C Occupied,Empty)}	State	3.2. Awake {(Normal Mode)/(C Occupied,Empty)}
SSD_28	3.2.1. Normal Mode {H CD Mode, Tuner Mode}	State	3.2.1. Normal Mode {H CD Mode, Tuner Mode}
SSD_53	<trigger><activity><state>	Transition	Trigger: Off Activity: Power down radio Next: Sleep
SSD_29	3.2.1.1. CD Mode Drive spins CD, delivers data stream.	State	3.2.1.1. CD Mode Drive spins CD, delivers data stream.
SSD_54	<trigger><activity><state>	Transition	Trigger: Tuner Activity: switch to tuner Next: Tuner Mode
SSD_55	<trigger><interface><state>	Transition	Trigger: Eject Activity: active α Next: Tuner Mode
SSD_30	3.2.1.2. Tuner Mode Radio plays signal received from tuner.	State	3.2.1.2. Tuner Mode Radio plays signal received from tuner.
SSD_61	<trigger><activity><state>	Transition	Trigger: Play CD (Occupied) Activity: Spin up CD Next: CD Mode
SSD_58	<trigger><activity><state>	Transition	Trigger: γ Activity: Spin-up CD Next: CD Mode
SSD_31	3.2.2. Eject Mode Radio is muted.	State	3.2.2. Eject Mode Radio is muted.
SSD_56	<trigger><activity><state>	Transition	Trigger: β Activity: Power down radio Next: Sleep

ID	System State Chart	Type	State Description
SSD_57	<trigger><activity><state>	Transition	Trigger: γ Activity: Power down radio Next: Sleep
SSD_50	<trigger><activity><state>	Transition	Trigger: On Activity: Power-up radio Next: Normal Mode {H CD Mode, Tuner Mode}
SSD_33	3.2.3. Safe CD is no danger in case of crash.	State	3.2.3. Safe CD is no danger in case of crash.
SSD_34	3.2.3.1. Occupied A CD is in the CD drive.	State	3.2.3.1. Occupied A CD is in the CD drive.
SSD_62	<trigger><activity><state>	Transition	Trigger: α Activity: Eject CD Next: Hangout
SSD_35	3.2.3.2. Empty No CD is in the CD drive.	State	3.2.3.2. Empty No CD is in the CD drive.
SSD_63	<trigger><interface><state>	Transition	Trigger: CD inserted Activity: active γ Next: Occupied
SSD_36	3.2.4. Hangout The CD has been ejected and can be removed.	State	3.2.4. Hangout The CD has been ejected and can be removed.
SSD_64	<trigger><interface><state>	Transition	Trigger: CD removed Activity: active β Next: Empty
SSD_65	<trigger><interface><state>	Transition	Trigger: Timeout Activity: active γ Next: Occupied

Figure 6, Textual representation of state chart.

A METHOD TO FIND ALL REQUIREMENTS

A Method for requirements completeness. The little story at the beginning shows, that overlooking a requirement can have severe consequences and sometimes the immediate financial consequences are the least of them. In (Carson '98) and (Carson '04) we find the statement *'Developing the complete set of requirements is equivalent to completely stating the problem to be solved.'* - or to put it the other way round: If we miss some requirements we develop a solution for the wrong problem. But how can we be sure that we have not missed some requirements? When is the set of requirements complete? What can we do to get a complete set of requirements? These questions have been asked before. A method that answer these questions we find in (Carson '98): *'The test for the completeness of the problem statement step is that all stakeholder interfaces are identified and quantified for all applicable development, operation, assembly, maintenance, and disposal phases and relating operating modes.'*

Completeness through use cases and state charts. The use case and state chart based method outlined above is a tool to implement the method to assure requirements completeness proposed in (Carson '98). The key is, to analyze the use cases thoroughly. For each use case all actors have to be identified and all interfaces these actors have with the system. Then, speaking with the words of (Carson '04), we have to quantify these interfaces. That is to define the possible states an interface (respectively the system behind the interface) can have and how it interacts with the actor in that state.

For instance in the state chart figure 3 and figure 6 we find the states *Safe* and *Hangout*. Where have they come from? The reason these two states exist is one of the interfaces the actor Passenger has with the CD-Player: the knee. In the case of a very rapid deceleration of the car, a passenger's knee may come into contact with the CD player's face plate. A CD sticking out of the CD slot can be hazardous in this situation. Thus when the CD is in the CD player the CD player-passenger-interface it is in a safe state but not while being partly ejected.

To find the complete set of requirements we start from the preconditions we have found when looking for the 'normal' functional requirements. Have we listed all preconditions? Very often the most obvious preconditions are not listed. For instance in the example figure 2 the fact is missing that the radio is mounted in a car. If we add these states to the state machine model and then answer the questions how did the system happen to enter this state, we find additional use cases, actors, and interfaces. To take up the example above, a use case

'Mount radio in model xy' gives us a wealth of interfaces and associated requirements like fastening points, electrical connectors, and points where a worker will exert pressure in order to push the radio into the desired position.

A system looked at as a black box and modeled as a state machine is defined by the states of its interfaces, because only the interfaces are visible to the outside world. This is exactly the information needed to ensure requirements completeness with the method proposed in (Carson '98).

As has been hinted above, there is still a reason to look not only at the interfaces but at the actors associated with the interfaces too. Often actors provide services to the system we develop. When we reduce the actors to their interfaces we codify their expected behavior as the protocol that governs the interface. But being external, totally independent entities there is no guarantee that they follow the protocol at all times. That is for full requirements completeness the case has to be considered, that the actor does not follow the protocol. For example in the use case 'Eject CD' above, success depends on the passenger to remove the CD while hanging out of the slot, but we also have to consider the case that the passenger does not follow the protocol and for example does nothing at all or even pushes back the CD into the slot.

Even more requirements. If the scope of the requirements analysis is not a strict black box view, there is a second reason to be aware of the actors and their capabilities. Not only the external entities may fail to provide the expected services but a system component may fail too. To provide the best possible service even under this circumstance we always should look whether an actor already involved in the use case may provide help. An example shall illustrate this aspect:

A car navigation system in order to provide its service needs the passenger to supply it with a data carrier, a CD or a DVD, that contains the geographic information like maps. While guiding the driver the situation may arise that a part of the data carrier is not readable. This happened to one of the authors in a rented car. The system responded to the situation with a message that it was not able to find the destination because the current location was outside the scope of the geographical information. This was of course not the case. After considerable puzzlement and then taking out the CD and removing a smear on the surface of the CD the system worked fine again.

If using our method and having captured the case, that a part of the data carrier is not readable, in the state chart, the developers of the navigation system had been driven to look for a use case that would bring the system from that error state back to the normal operational state. Is there an actor already involved that can provide help? In the example above a message to the passenger, to remove the CD and to clean off eventual dirt from the data carrier because there are read errors, would have been an excellent solution.

The combination of use cases and state charts supports requirements completeness with respect to exceptional states of external entities but also lets us find solutions to recover from exceptional situations within the system.

CONCLUSIONS AND FURTHER DEVELOPMENT

A method that allows broad participation in the requirements validation and verification. The combination of use cases and state charts as requirements notation has proved to have many advantages. Both models capture essentially the same information but with a different focus. Use cases present the information in a more easy to understand form that is but not mathematically exact. State charts present the state machine model of the requirements and thus inherit the mathematical foundation of state machines. By using a requirements management tool we can generate both views, the use case representation and the state chart representation, from one set of data. This enables us to communicate the requirements in the database to a broad range of stakeholders in an appropriate form. Less

technically oriented persons easily grasp the gross picture from the use case descriptions and the technical experts find the required level of detail in the combination of use cases and state charts. This helps to get a thorough review of the requirements.

Replacing actors with their interfaces helps the state model. While traditionally the activities described by a use case are between the system and external entities called actors, it is of advantage to replace the actors with the interfaces through which the actors interact with the system. The states and state changes of these interfaces during the course of service delivery are modeled with state machines and represented by state charts. Iterating back and forth between use cases and state charts very quickly fills gaps in the requirement information.

Analysis of interfaces provides the non functional requirements. For each interface there exists a protocol, that is there are rules that describe how the system and the external entity can and will interact. From the analysis of the modes of interaction we learn the possible interface states and the required performance parameter values (how fast has the response to be, how strong can be the impact, at what frequency will be the radiation ...).

Requirements completeness. Our method of requirements analysis is complementary to the approach described by (Carson '98) to achieve requirements completeness.

Further development. To apply our method efficiently on complex systems we must be able to partition the task of analyzing the system's requirements. Already use cases can be analyzed in parallel. What we need is a way to define sub-systems to work on them in parallel. While the state chart notation is designed to support sub-systems the use case notation in its usual form is not. Our next step is to modify the methodology to incorporate the partitioning of the system and to enhance the notation of use case sequences in a way that supports composing and decomposing of use cases.

REFERENCES

Carson, Ronald S., "Requirements Completeness: A Deterministic Approach", Proceedings of the 8th International Symposium of the International Council on Systems Engineering, 1998, Vancouver, Canada.

Carson, Ronald S. et. al. "Requirements Completeness", Proceedings of the 14th International Symposium of the International Council on Systems Engineering, 2004, Toulouse, France

Harel, David, "Statecharts: A Visual Formalism for Complex Systems, *Science of Computer Programming 8*, pp. 231-274, 1987, Elsevier Science Publishers B.V. (North Holland)

Kaffenberger, Ruediger, "Requirements Engineering with Goal Driven Use cases", Proceedings of the 12th International Symposium of the International Council on Systems Engineering, 2001, Las Vegas, USA

OMG, *Unified Modeling Language Specification*, Version 1.3. Object Management Group, Inc., June 1999. http://www.rational.com/uml/resources/documentation/index.jsp

OMG, *Unified Modeling Language: Superstructure*, Version 2.0, Revised Final Adopted Specification (ptc/04-10-02), Object Management Group, Inc., October 8, 2004

BIOGRAPHY

Ruediger Kaffenberger has earned a Dipl.-Ing. degree in electrical engineering from the University of Stuttgart, Germany. He designed software and hardware for single-board computers for the control of telecommunication systems. Since 1992 Ruediger is working as a systems engineer, mainly in the field of requirements engineering. After nearly 20 years in the telecommunications industry he now works mostly in the automotive sector.
Ruediger is a member of the IEEE, a founding member of the INCOSE German Chapter

(GfSE), and treasurer of GfSE.

André Zeh has studied electrical engineering in Chemnitz and Moscow and earned a Dipl.-Ing. degree from the Technical University of Chemnitz, Germany. He worked in software development and architecture in the automotive sector and for safety critical systems. Since 1997 he is dealing with software and systems engineering. André is certified ISO/IEC 15504 (SPICE) assessor and supports manufacturers in the improvement of their development processes.