Combining Test Case Generation for Component and Integration Testing

Sebastian Benz
BMW Car IT GmbH
Petuelring 116
80809 Munich, Germany
Sebastian.Benz@bmw-carit.de

ABSTRACT

When integrating different system components, the interaction between different features is often error prone. Typically errors occur on interruption, concurrency or disabling/ enabling between different features. Test case generation for integration testing struggles with two problems: the large state space and that these critical relationships are often not explicitly modeled. The approach presented in this paper is to use task models to describe the interaction between environment and system. This restricts the possible state space to a feasible size and enables the generation of task sequences, which cover the critical interaction scenarios. These task sequences are too abstract for testing the System-Under-Test (SUT), due to missing input- and expected output behavior. To overcome the different abstraction levels, the tasks are mapped to component behavior models. Based on this mapping, task sequences can be enriched with additional information from the component models and thereby executed to test the SUT.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Validation; D.2.5 [**Testing and Debugging**]: Testing tools

General Terms

Model Based Testing, Integration Testing

Keywords

Model Based Testing, Verification, Automatic Test Case Generation, Integration Testing

1. INTRODUCTION

Testing is the most often used mean of verification. During the development of a system, 50 percent of the time and more than 50 percent of the total cost are expended for testing [24]. Model based test case generation is a promising approach to reduce the effort of test case creation and enables the systematic selection of test cases. Furthermore in combination with an automated execution of test cases the automatic generation of test cases allows a continuous testing process during the development of a system. Especially for systems with different configurations and region specific variants, such as an automotive infotainment system, the automatic generation of test cases is necessary, because each variant requires its own test suite.

The basic principle of model based test case generation is to derive test cases from a behavior model based on certain selection criteria. Such a behavior model is herein referred to as a test model. The major drawback of model based testing, is that often the state space of the system is very large, which leads to a large number of potential test cases. As a result the full coverage of a system's behavior by automatically generated test cases is not feasible. A human tester struggles with the same problems, but based on his experience he implicitly abstracts from the uncritical system properties and reduces thereby the number of test cases.

For that reason model based testing is always connected with some kind of abstraction [23]. This is necessary, because otherwise the effort for validating the test model is identical to the effort for validating the implementation. Furthermore abstraction narrows the available state space, which reduces the possible number of test cases. As a result of abstraction the generated test cases are abstract test cases, which have to be enhanced with additional information to enable an execution against the System Under Test (SUT). This is called test case instantiation [22] and is either performed by a translator, which adds the missing information (e.g. mapping from equivalence classes to concrete data, to overcome data abstraction) or by driver components, which encapsulate missing information (e.g. an abstract user input is replaced by a driver component, which performs the input). The risk of abstraction is that potential errors are abstracted away. Ideally abstraction omits all dispensable details and forces the focus on error prone system properties. As a consequence a good choice of the abstraction depends on the domain and properties of the system. For example communication abstraction for testing a communication protocol is absurd. But for testing the Graphical User Interface of a system, communication abstraction is the right approach.

Furthermore the focus of testing, and thereby of abstraction, depends on the current development phase. The main principle in the development of a large software system is divide and conquer. First the system is divided into its components and these are further refined and implemented. After implementation, these components are successively composed to form the whole system. Each development step is accompanied by testing. Therefore the generation of test cases for each of the three phases of testing (component, integration, and system testing) is desirable. During component testing the focus lies on the interface-behavior and internal-behavior of a component. In contrast, during integration testing and system testing the interaction between components is of particular importance. Typical error prone scenarios are interruption or disabling of a feature by other features, due to the shared usage of resources such as input and output devices. The focus on the interaction and the abstraction from the component-specific behavior leads to a higher degree of abstraction of the integration test model, in comparison to the components' test models. As a consequence for the execution of integration tests, more information has to be added to overcome the gap between abstract test model and SUT.

The contribution of this paper is a method for interaction modeling that covers the typical error prone interaction scenarios. These scenarios are modeled using task models, which describe the tasks the system is able to perform in interaction with the environment. Task models abstract from the component behavior and inter-component communication and describe explicitly the temporal relationships between different tasks. This leads to a reduction of the state space and enables an effective generation of test cases. Due to the higher abstraction of the interaction model the generated test cases require more additional information for an execution than component test cases. The approach presented in this paper is to avoid the effort of implementing additional driver components for interaction testing by reusing component models and test generators for the generation of test cases, as shown in Figure 1.

The approach is demonstrated for a case study from the automotive infotainment domain, that is introduced in the next chapter. However the approach is not restricted to the automotive domain and can be applied to other domains where the integration of different features in one system is error prone, such as mobile phones or telecommunication systems.

2. CASE STUDY

The case study is a scenario from the automotive infotainment domain. An infotainment system is a distributed system with several Electronic Control Units (ECU), which communicate over a bus system, such as the MOST bus. The case study consists of a telephone application which allows the user to manually enter a telephone number or select an existing number from the address book. Additionally there is a CD Player application, that allows the user to listen to CDs. The CD Player is able to start, stop and pause the playback of a CD, which is inserted or ejected by the user.

Both applications can be controlled by a Graphical User Interface (GUI) and a control knob. With the control knob the user can scroll up and down in a list and activate a selected button by pressing the knob. An additional menu button allows the user to navigate to the parent menu.

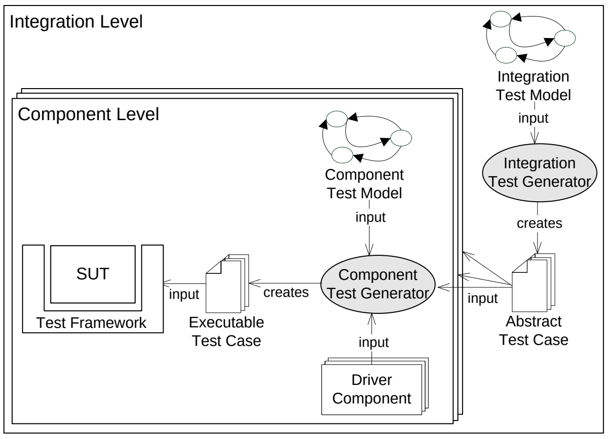


Figure 1: During component testing the test cases are created based on the component models. These are executed using driver components. The abstract test cases created during integration testing are further enriched with additional information from the component models before execution.

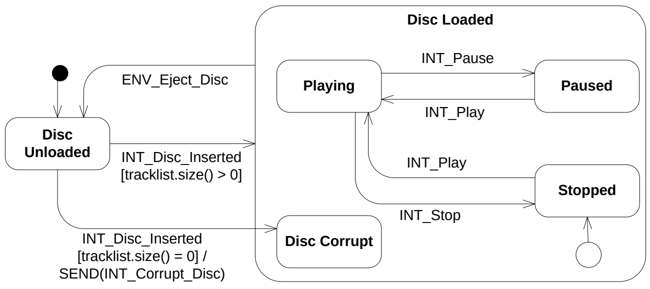

Disc Loaded
ENV_Eject_Disc
Playing
INT_Pause
Paused
INT_Play
Disc Unloaded
INT_Disc_Inserted [tracklist.size() > 0]
INT_Play
Stopped
INT_Stop
INT_Disc_Inserted [tracklist.size() = 0] / SEND(INT_Corrupt_Disc)
Disc Corrupt

Figure 2: The Statechart Model for the CD Player component.

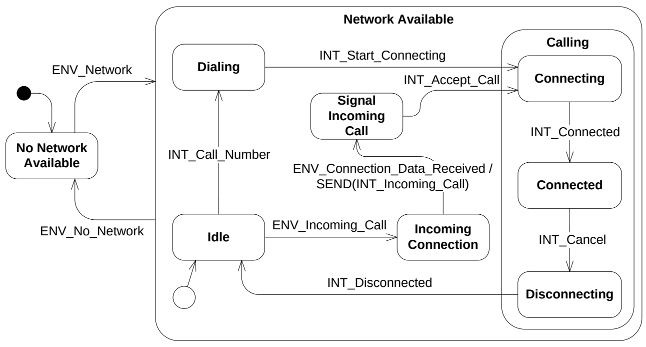


Figure 3: The Statechart Model for the Telephone component.

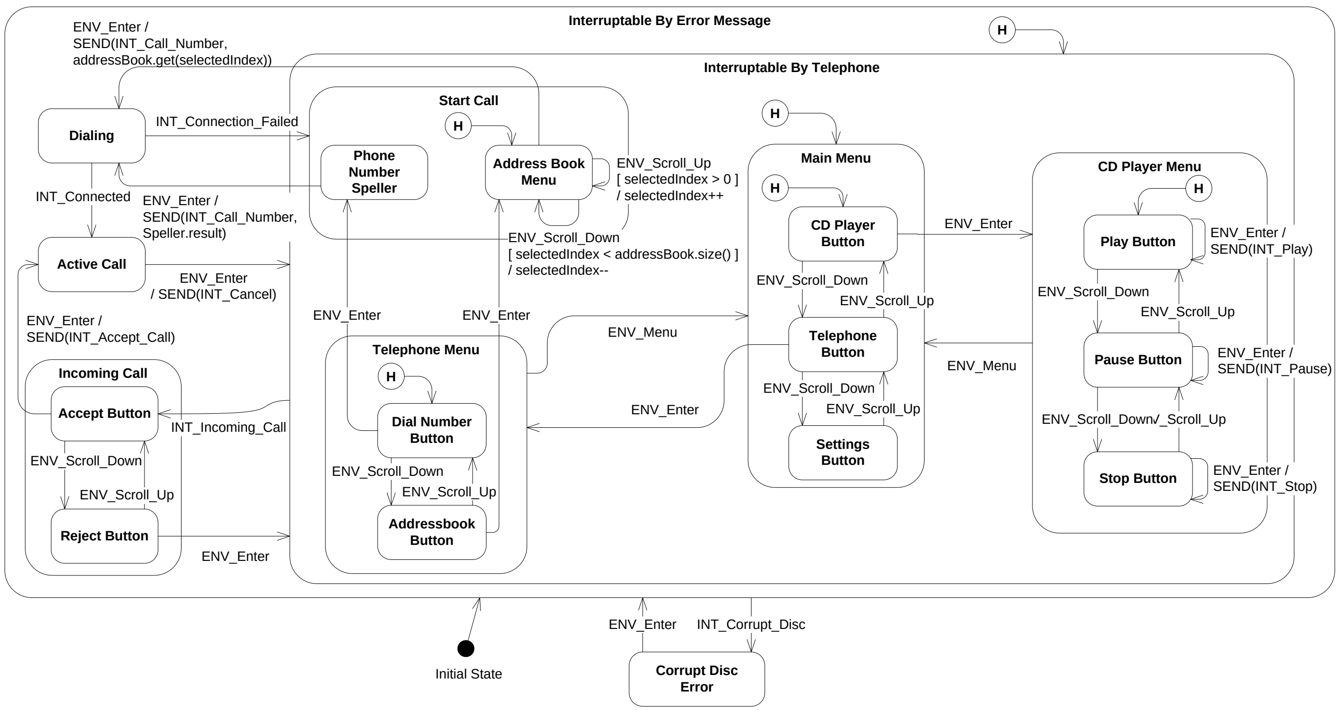

Interruptable By Error Message
Interruptable By Telephone
ENV_Enter / SEND(INT_Call_Number, addressBook.get(selectedIndex))
H
Dialing
INT_Connection_Failed
Start Call
Phone Number Speller
Address Book Menu
ENV_Scroll_Up [selectedIndex > 0] / selectedIndex++
ENV_Scroll_Down [selectedIndex < addressBook.size()] / selectedIndex--
INT_Connected
ENV_Enter / SEND(INT_Call_Number, Speller.result)
Active Call
ENV_Enter / SEND(INT_Cancel)
ENV_Enter / SEND(INT_Accept_Call)
Incoming Call
Accept Button
INT_Incoming_Call
ENV_Scroll_Down
ENV_Scroll_Up
Reject Button
ENV_Enter
Telephone Menu
Dial Number Button
Addressbook Button
ENV_Menu
Main Menu
CD Player Button
Telephone Button
Settings Button
CD Player Menu
Play Button
ENV_Enter / SEND(INT_Play)
Pause Button
ENV_Enter / SEND(INT_Pause)
Stop Button
ENV_Enter / SEND(INT_Stop)
ENV_Scroll_Down/_Scroll_Up
Initial State
INT_Corrupt_Disc
Corrupt Disc Error

Figure 4: The Statechart Model for the GUI component.

There is a functional specification in form of a complete State Chart model, which describes the functional aspects of the system. The system itself is distributed in three ECUs, one for each software component, which communicate using a communication bus. The software components are the Telephone, the CD Player and the GUI. Figures 2, 3 and 4 show the behavior models for the applications and the GUI. For purposes of clarity, the models abstract from the concrete bus communication protocol and from the concrete input and output devices. The name of events triggered by the environment starts with "ENV_". Internal events' names start with "INT_". The GUI model describes only the dialog between user and system, based on abstract graphical input elements, like lists and buttons.

3. APPROACH

The root of the matter is the question at which abstraction level the interaction between different components should be described. In practice different component models are composed at the communication level. For the generation of test cases this is the wrong level of detail because it corresponds to the implementation level and results in a large state space. Furthermore it is difficult to determine which message sequences leads to an error prone interaction scenario (e.g. an interruption signal) and which messages are uncritical (e.g. status messages). The reason lies in the fact, that this information is not available at the component level.

For the decision about the right abstraction level, a typical system architecture has to be considered. There are three kinds of system components:

Feature Components: encapsulate certain features, which form the system characteristic (e.g. CD Player, Telephone).

Boundary Components: link the system to the environment (e.g. User Interface, control knob). They represent the composition of all system functions to the environment.

Supporting Components: provide functionalities or resources which are required by other components to realize their behavior (e.g. Audio Manager, which switches audio channels between telephone and CD Player). The components and their functionalities are not visible to the environment.

These components are normally identified during the early design phase and are further refined, implemented and tested. The component specific behavior models, which are created during the development are further referred to as component models.

Error prone situations are interactions between different features. Each feature is often implemented independently of the other features in its own functional component. Possible interactions between features are interruption, concurrency, resuming and disabling. These interactions are often initiated by the environment and therefore the point of interruption is unpredictable. The problem is that *Boundary Components* and *Supporting Components* have to react to all possible interaction scenarios and have to switch between different feature contexts. For instance they have to configure input and output devices depending on the active feature.

The possible interaction scenarios are often not explicitly modeled or specified, and are implicitly encoded in the *Boundary Components* and *Supporting Components*. For the purpose of testing, a model is required, which describes explicitly the relations between different features. The interaction between environment and system is defined by the activities the system is able to perform at a certain point in time and which consequence the execution of an activity has on the other activities. If a model describes this information, the generation of test cases for critical changes between activities will be possible.

Task Models offer an appropriate abstraction level and describe exactly the described above critical interaction scenarios. They are used in the early phase of model based User Interface (UI) development to describe the possible tasks the user can perform in interaction with the system. The UI of a system represents the composition of all system's features. Therefore the UI must know the relationships between the different features. For example, if an incoming call can interrupt the playback of a CD, the GUI dialog must provide a transition from the CD Player screen to the screen, that signals the incoming call. In the model based development of UIs task models are used to model these situations. A task is defined by a certain goal, which is reached by an activity. For instance the goal is to make a phone call and the activity is to dial a number. A task model contains the different tasks and subtasks a user can perform during his interaction with the system. It describes the relationship between the different functions from the users point of view, like interruption, disabling or enabling. These are exactly the typical error prone situations, which should be tested during integration testing. In addition task models abstract from the functions' implementations and the functions' representations (e.g. GUI Dialog), which leads to a reduction of the required state space.

For this reason the basic approach is to use this modeling concept from the UI development for the purpose of testing. However, for testing purposes the system must be described more generally from the environment's point of view. In the context of test case generation the ConcurTaskTree (CTT) Notation [18] is suitable, because it offers a hierarchical structuring of tasks and provides a formal definition of the tasks temporal relations. A task can be divided in several subtasks. For example the task "CD Playback" can be divided in the subtasks "Play" and "Stop". Based on the task hierarchy the temporal relationships between tasks are defined. The temporal operators, like "Enable" or "Suspend/Resume" are taken from the Language of Temporal Ordering Specification (LOTOS) [2]. For example the task "Incoming Call" is related by the "Suspend/Resume" operator with the task "CD Playback". This implies, that an incoming call suspends all subtasks of the task "CD Playback". Using this notation, all possible interaction scenarios can be defined.

A further advantage is, that task models describe the relations between different features, as well as the relations between different feature-specific tasks. Therefore test cases can be generated for testing not only the interaction between different features, but also the feature specific usage scenarios.

With the application of task models, new task based test selection criteria are possible. These selection criteria focus on usage scenarios and provide a new possibility to measure

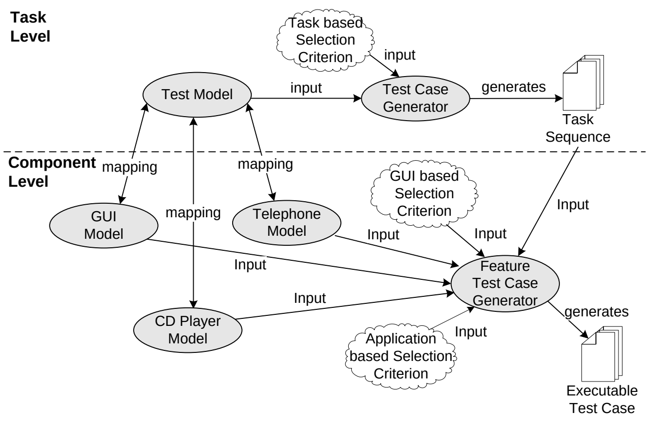


Figure 5: Each task's preconditions and postconditions in the test model are mapped to corresponding states in the component models for the GUI, the CD Player and the Telephone. Based on task specific selection criteria (e.g. Task Coverage), task sequences are generated. Using the task-component-mapping, these abstract task sequences are instantiated using additional information about input behavior and output behavior from the component models, to create executable test cases.

the tested aspects of a system. Based on this coverage criteria task sequences are generated. Each of them represents a possible test case. The problem is, that such generated test cases are too abstract for execution, because of the test's missing input and output behavior.

To solve this problem each task will be mapped onto the existing component models. The mapping is performed by describing each task's preconditions and postconditions by constraints on the underlying component models. With the information from the mapping it is possible to derive a task's input behavior and expected output behavior from the component models and therefore enables the transformation of a task sequence to an executable test case.

One might argue that these test cases are still too abstract because they cover only the main usage scenarios. But by generating test sequences based on the component models, additional component model specific coverage criteria could be applied. So for one task sequence a complete test suite can be generated, depending on the component specific test coverage criterion. Figure 5 shows the generation of test cases based on the test model at the task level. The combination of test case generation on task level with test case generation at component level is called "Coupled Test Case Generation".

The proposed approach applies especially for automotive infotainment systems, because they combine different features like navigation, communication and entertainment, which share the input and output modalities (GUI, speech, haptic interfaces). But in general this concerns all systems that integrate different features that interact indirectly by using the same resources and where the activation of a feature is influenced by the environment. In summary the approach can be applied to event based, reactive systems that integrate different features and are characterized by a strong interaction with their environment.

4. THE TEST MODEL

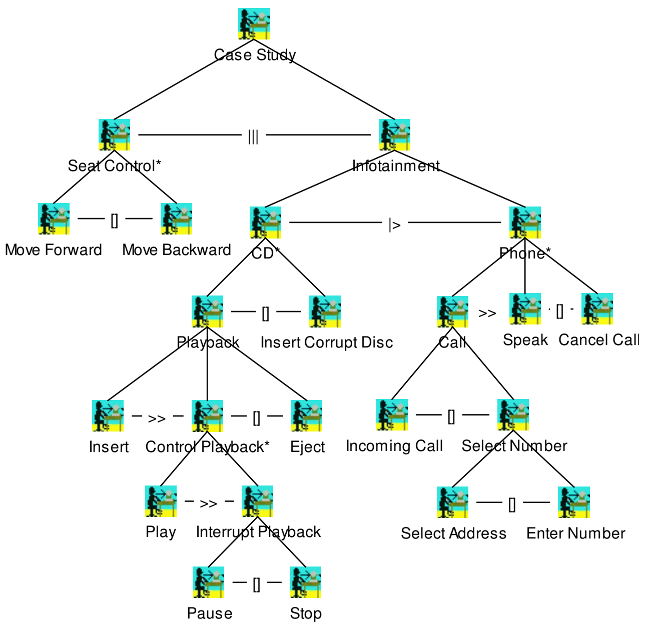


Figure 6: The task model of the case study specified with CTTE.

A task defines how the user can reach a goal in a specific application domain. The goal is a desired modification of the state of a system or a query to it [19]. Task models are used in cognitive science to characterize and identify tasks in the early requirements phase [12, 26]. In the model based user interface development task models are used for an early formal description of tasks and their relationships, examples are GOMS [11], MDL [25], UAN [9] or the ConcurTaskTree Notation [18].

The difference between task models and object oriented methods such as UML [16] is that object oriented approaches focus on modeling the objects, composing the system and the interaction between them. Task models describe activities and the relations between activities [18].

4.1 The ConcurTaskTree Notation

For specifying the interaction between the system components, in the context of testing, the ConcurTaskTree Notation (CTT) is appropriate. The reason is the formal definition of temporal relations, which allows a formal analysis of the task dependencies. Several approaches are using CTTs for formal validation and model checking [20, 1] of user interfaces.

Another advantage of CTTs is the compact notation, which is easily understandable by users without a formal background. With CTTs even large systems can be graphically described, because they focus only on one aspect, in contrast to other graphical modeling notation, like State Charts or Activity Diagrams. For example in [15] CTTs are mapped to UML 2.0 Activity diagrams, which resulted in incomprehensible large diagrams.

Figure 6 shows the task model for the case study, which was created using the CTT modeling environment CTTE [14]. In the succeeding part of this section the CTT are described more detailed on the basis of the case study. There are four types of task in a CTT:

User Tasks: are entirely performed by the user.

Application Tasks: are completely executed by the system.

Interaction Tasks: are performed by the user interacting with the system.

Abstract Tasks: are complex tasks, which are divided in different subtasks.

The set of all tasks is a tree, where a task is divided in its subtasks starting from a root task. The temporal relations are defined between two sibling task and applies for all subtasks. For example the task "Phone" in the case study suspends all subtasks of the "CD" task. In detail there are the following kinds of temporal relations:

Choice ([]): It is possible to choose between a set of tasks. If one task is active the other tasks from the set could not be selected. For example the user can stop or pause the playback.

Order Independence ($|=|$): Both tasks have to be performed, but if one task has started, it has to be finished, before the other can start.

Concurrent (|||): Both tasks can be executed concurrently.

Disabling ([>): When one task is finished, it disables an other task. For example if a disc is ejected, the task "play disc" is disabled.

Suspend/Resume (| >>): The execution of a task $t1$ interrupts the execution of another task $t2$, which resumes after $t1$ has finished.

Enabling (>>): If a task finishes, it enables the execution of another task. For example the task "insert disc" enables the task "play".

Iteration ($*$): The task can be performed multiple times.

4.2 Using ConcurTaskTrees for test case generation

Goal of CTTs in the UI development is to describe the tasks the user performs while interacting with the system. This includes tasks the user performs without interacting with the system (the User Tasks). For test case generation purposes only tasks, which include an interaction between system and environment (which includes the user) are relevant. Examples for environment tasks are "insert disc", which is performed by the user and "incoming call", which is initiated by the environment. The focus moves from a user centric task model in the UI development to a system centric task model for test case generation.

Currently a task model describes only the positive tasks. But especially scenarios where invalid inputs occur are important for testing a system. To describe these scenarios a new task category "error task" is necessary. This concept contradicts the original task definition, because the user's goal is not to cause an exception. Otherwise speaking from a testers point of view, the goal is to provoke erroneous behavior. The task model in Figure 6 contains all tasks of the case study. An example for an "error task" is the task "Insert Corrupt Disc". Based on the temporal relationships between the tasks all valid task sequences are defined. Each task is a potential test case, whereas the activity corresponds to the input behavior and the goal to the expected output of a test case. In the context of integration testing, not only a task is a test but in particular a sequence of different tasks is a potential test case, because the change between two tasks is often linked with a critical context change. An example is the task sequence "Insert Disc" → "Play" → "Incoming Call". Critical in this sequence is the change between the CD Player context and the Telephone context.

A task defined by a CTT is missing two elementary characteristics of a test case, a formal definition of the input behavior and the expected output behavior. An obvious approach would be assigning each task with an input and an expected output. For example the task "Insert Disc" has as input "Input_Insert_Disc" which would be assigned to a test driver. This commands a test roboter to insert a disc. The expected output would be, that the GUI shows a playing icon, which could be determined by a screen grabber. This might be a possible solution for some tasks. The problem occurs, when a task's input depends on the history of executed tasks. Consider the task "Play": if the last active task was "Speak", the input sequence to trigger the task "Play" would be completely different from the input sequence, if the last task was "Stop", because the GUI dialog state depends on the previous task. Hence for each possible previous task a special input sequence must be specified, which means additional modeling effort. Furthermore this results in a redundant description of the system, because all necessary information is contained in the component models, like the GUI model in the case study.

The solution is to map the tasks to the component models. Based on this mapping the input sequence for a test case can be generated from the more detailed component models. How the mapping can be accomplished is matter of the next section.

5. BRIDGING THE GAP BETWEEN TEST MODEL AND COMPONENT MODELS

The common definition of a task is "*an activity performed to reach a certain goal*". A test case is defined as a structure of input behavior and expected output behavior [3]. In connection with the definition of a goal as a "*a desired state in the system or task world*", the mapping from a task to test case is intuitive. The goal of a task corresponds to the expected output of a test case. A desired state in the system is described by a set of component states. For example the goal of the task "Incoming Call" would be that the telephone component is in the state "Incoming Call" and the GUI is in the state "Accept Button", which is a substate of the composite state "Incoming Call".

The test's input behavior depends on the activity, that has to be performed to reach the goal. This activity is characterized by an interaction between the environment and the system. Therefore the test's actual inputs are the inputs performed by the environment during a task's activity. There are two scenarios for the interaction between environment and system:

Single Trigger: A task's activity is initiated by a single input from the environment, which triggers the activity. For example the trigger for the task "insert disc" is the insertion of a disc by the user. The remaining behavior of the activity is performed by the CD Player component, like reading the disc and notifying the GUI. The mapping between task and test case is simple: namely the event "ENV_Insert_Disc" is the input for the test case. To enable a later test case instantiation from a task, the task must be assigned the corresponding input event from a component model. If the trigger can be performed by different components or input devices, several triggers can be assigned to a task.

Trigger Sequence: The activity is performed by several triggers from the environment. For example entering a telephone number using a speller is performed by entering each digit separately, which is connected with several user inputs. In this case only the initiating trigger must be assigned to the task. The missing triggers can be generated from the corresponding component model, by using the precondition and postcondition of the task. The exact procedure will be described in the next section.

Furthermore there must be differentiated between state dependent and state independent triggers. In the case of state dependent triggers, an input event is linked to a state in the model. For example all triggers by the control knob depend on the current state of the GUI dialog. For instance in order to start the playback of a CD, the play button must be selected, when the control knob is pressed. In this case the trigger must be additionally linked to a precondition defined on the component model.

To enable the generation of test cases based on a task sequence, the relevant preconditions of a task must be defined based on the component models. For example the task "Play" requires that the play button is selected. This is important if the postcondition of the previous task differs from the precondition of the next task and therefore additional inputs have to be generated to establish the preconditions. Moreover a defined precondition guarantees, that the test is always performed on the same preliminaries and the test's verdict has therefore the same significance for different test executions.

Only the leaf tasks of a task model are executed. The parent tasks are finished, depending on their child tasks. As a consequence a trigger can only be assigned to leaf tasks. The preconditions and postconditions defined for a parent task apply to the child tasks. The parent task's precondition must be fulfilled before the first execution of a child task. For example the task "Seat Control" in the case study has the precondition that the car is stopped. This condition must be applied to both child tasks, because each can be executed first, due to the *Choice Operator*. The same applies

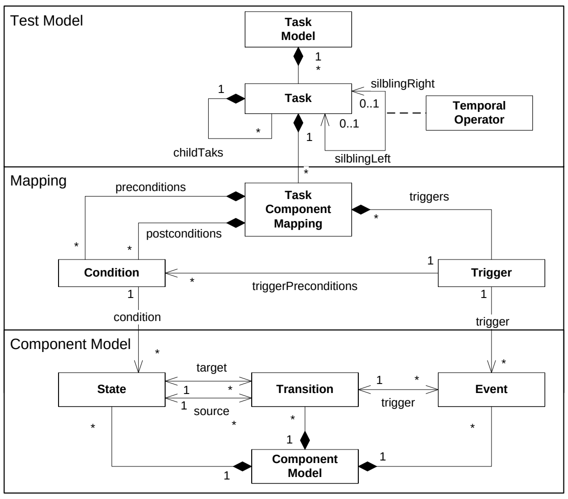


Figure 7: The mapping between task model and component model.

to the postcondition due to the *Choice Operator*, the parent task "Seat Control" is finished after one of the child tasks is finished. Figure 7 shows the resulting mapping between task and component model.

6. COUPLED TEST CASE GENERATION

The creation of executable test cases is performed in two steps. First the task sequences are created based on task specific selection criteria or manual task selection. The goal of this step is to find test cases, that cover the error prone interaction scenarios. These scenarios are described by the selected task sequences. In the second step the sequences are translated into executable test cases by replacing the tasks by concrete system inputs and expected system. The translation is based on the task–component-model mapping described in the last section.

6.1 Deriving Test Cases from Task Models

A task model formally describes the different tasks and subtasks, that can be performed by the system. The advantage of the CTT Notation is the formally defined temporal relationships between tasks. Based on these relationships all possible sequences of tasks are described. Normally a system consists of iterative tasks, therefore the number of possible task sequences is infinite. As a consequence there are selection criteria necessary, which limit the number of task sequences to a finite set.

There are several well-established approaches for test case selection. Typical are coverage criteria (e.g. State Coverage, Transition Coverage in a State Transition Network) [17], random test case selection, stochastic test case selection [27]. To the best of the author's knowledge there are currently no approaches for test selection criteria in the context of task models.

The criteria proposed in this paper are combinations of different techniques. To some extent classical coverage criteria are applied to task models, but additionally specific test selection criteria based on the characteristics of a task model are presented. Selection criteria for the generation of task sequences are:

Manual Selection: The manual selection of test cases is still important. The selection is based on feedback from the field, the tester's experience or the systematic analysis of specific scenarios and is an extension to the automatic generation of test cases.

Task Coverage: Each task is performed once. This is a simple coverage criterion which assures, that every task can be performed. The significance of this criterion is moderate, because it makes no assumption regarding the interaction between different tasks.

Simple Temporal Relationship Coverage: Each specified temporal relationship is executed once.

Complete Temporal Relationship Coverage: Each temporal relationship is executed for all possible task combinations. This criterion covers all possible interaction scenarios between different feature and all feature specific activity sequences.

n-**Task Combination Coverage:** Each possible sequence of *n*-Tasks is performed.

Interruption Coverage: Each possible interruption of a task by another task is performed.

Random Selection: Random selection of task sequences.

One extension of the above stated criteria is necessary for concurrent tasks. When tasks are executed concurrently, one task can be active during the parallel execution of several other tasks. For example, while the user is moving his seat forward, he inserts a disc and activates its playback. The task "Move Forward" starts before the task "Incoming Call" and finishes after the task "Speak" has finished. These situations are critical, especially if some *boundary components* or *supporting components* are used by both tasks. Concurrent task behavior cannot be described by a task sequence consisting of tasks and their temporal relationship. To solve this problem, a task has to be described in a task sequence by its starting point and its endpoint. Therefore from the start and the end of a task, it can be determined, which tasks are executed concurrently. As a consequence a new test selection criterion to restrict the number of possible task sequences has to be defined:

n-*m*-**Concurrency-Coverage:** The maximum number of *n* concurrent task sequences is defined by *n*, and *m* restricts the number of sequential tasks performed during the execution of additional, concurrent "task threads". When using 2-1-Concurrency-Coverage the situation above would not be tested, because the execution of two sequential, concurrent tasks is not regarded. But the 2-2-Concurrency-Coverage criterion covers this situation. Figure 8 shows two task sequences. Both contain the same tasks, but differ in the starting point of the concurrent task "Move Forward". The first sequence is an example, which is covered by 2-1-Concurrency-Coverage: during the execution of the task "Speak" is only the task "Move Forward" concurrently executed. Whereas the second sequence is covered by 2-2-Concurrency-Coverage, because during the execution of "Move Forward" are the two tasks "Incoming Call" and "Speak" sequentially performed.

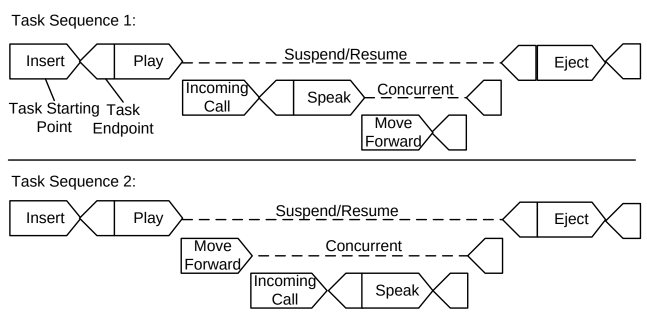


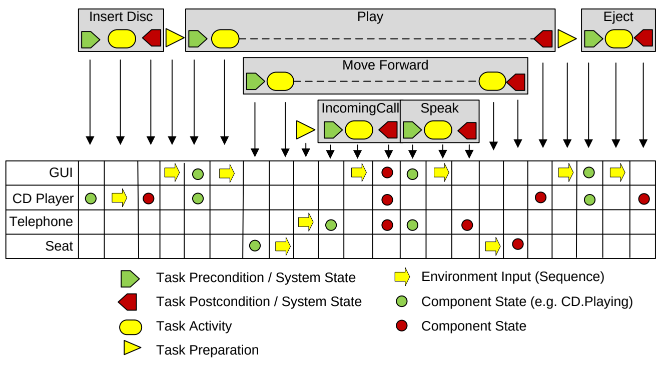
Figure 8: Two task sequence with suspend/resume and concurrent task relations. Each task is described by its starting point and endpoint.

A similar problem exists for task interruption. For instance the task "Play" is suspended by "Incoming Call" and "Speak". So the task "Play" is finished, after the two interrupting tasks. But in contrary to concurrent tasks, this is explicitly defined by the hierarchical structure of the CTT. "Play" resumes after the *abstract task* "Phone" has finished. Nevertheless by describing a task in a sequence by its starting point and ending point, as shown in Figure 8, this information must not be taken from the task model and is included in the sequence.

6.2 Using Component Models for instantiating a task sequence

Each task represents a possible set of test cases. Based on the task component mapping the derivation of an executable test case from a task definition is straight forward. A test requires as starting point a defined system state, to ensure the repeatability of the test case. Such a system state is defined by the task's preconditions, which cover all by the execution of the task affected component states. The input behavior of the test case is defined by the task' triggers and the expected output by the task's postconditions. The question is, how can a sequence of tasks be mapped to an executable test case, based on the temporal relations between the tasks and the task-component mapping. The basic approach is to define a test case by a sequence of system states, which the system has to reach during the execution of the test. The task's starting and ending points, defined in the task sequence, represent these system states. Each system state corresponds to a set component states, which are defined in the task-component mapping. The result is a chain of component states, which can be easily retrieved from a given task sequence.

Still missing is the test's input behavior. This can be generated based on the component models by finding a sequence of environment inputs, that change a system state into another system state. This must be applied twice: when a component's postcondition differs after fulfillment of a task from the component's precondition of the next task and for the actual activity of a task. Figure 9 shows the



Insert Disc
Play
Eject
Move Forward
IncomingCall
Speak
GUI
CD Player
Telephone
Seat
Task Precondition / System State
Task Postcondition / System State
Task Activity
Task Preparation
Environment Input (Sequence)
Component State (e.g. CD.Playing)
Component State

Figure 9: The mapping between task sequence and component models. The sequence covers a typical scenario. The user inserts a disc and starts the playback. While listening, he adjusts his seat. The listening is paused by an incoming call, which he finishes, after a short conversation, to stop the adjustment of his seat. After the call is finished, the playback resumes.

task sequence from the last section. Each task is divided in its precondition, postcondition and activity, which are mapped to the the corresponding components. For instance, the precondition for the task "Play" is defined in the GUI (GUI.Play_Button) and the CD Player (CD.Stopped OR CD.Paused) component. The activity of a task is mapped to an component specific input sequence, which is performed by the environment, e.g. "ENV_Enter" to activate the play button. Furthermore the figure contains the task preparations. This symbolizes, that additional input is necessary, because a component's postcondition differs from the component's precondition of the next task. The task preparation is also mapped to a component specific input sequence, for instance the necessary navigation steps to reach the play button before the task "Play" can be performed.

The composition of the task test cases based on their preconditions and postconditions can be applied to sequential tasks, but for temporal operators like *suspend/resume* and *concurrent* the sequential composition is not feasible, due to the necessary interweaving of the tasks' input sequences. For example if the task "Move Forward" is executed concurrently to the tasks "Incoming Call" and "Speak", the triggers "ENV_Start_Forward" and "ENV_Stop_Forward" can be performed anytime between the triggers of the concurrent tasks, which leads to a large number possible serializations of the different input sequences. Therefore new test selection criteria are necessary to restrict the number of generated input sequences for the *suspend/resume* operator and the *concurrent* operator.

Concurrent: The concurrent execution of tasks leads to a large number of possible input sequences, due to the arbitrary execution order of the concurrent tasks. Possible alternative selection criteria to the full input sequence coverage are random or alternating input ordering.

Suspend/Resume: A task can be interrupted at different points during its execution. Under the assumption, that no timed models are available, one can only differentiate between inputs and component states after which the interruption can occur. In the case study, the incoming call can interrupt the "Play" task before or after the "Play Button" is pressed. Test selection criteria can be applied for *Suspend/Resume* as well. Possible candidates are full interruption coverage or random interruption.

The basic principle of generating the input sequences is the automatic derivation of a path in a state machine between two given states. When applying this to a distributed system, like the case study, the problem occurs, that the input sequence has to be generated starting from a set of distributed states and the target is defined by several distributed states as well. Depending on the size of the system, this search can be very time consuming. To solve this problem AI planning methods can be used. The component models have to be transformed in a formal language such as STRIPS [6], which has been applied for State Charts in [7]. A STRIPS specification consists of a set of a description of the initial states, a description of the desired goal, and a set of possible actions. Based on this a planner produces a sequence of actions that lead from the initial state to a state meeting the goal, which is the required input sequence. There exist different approaches, which use planning for test case generation [10][8].

The input behavior of a test case consists only of environment inputs. In association with the fact that tasks are feature specific, it can be safely assumed that an input sequence can be derived from one component model, without regarding the other components. This implies that for the execution of a task only one *Feature Component* is triggered by the environment. The basic assumption is that each task covers a certain functionality, which is implemented in one of the components. Tasks describe only the interaction between system and environment and not the interaction between different components. The execution of a task can influence the state of other components, such as the telephone which pauses the CD Player when an incoming call arrives. But this is internal system behavior and must not be triggered by a test case. The result of this action, the paused CD Player, is an expected output behavior, but this is specified by the task's postcondition and must therefore not be generated from the component models. When a task can be performed using different *Boundary Components*, for example, if the user can initiate a telephone call either by a speech interface or the GUI, the restriction must be made that one task can only be performed by one *Boundary Component*. However, the execution of two successive tasks using different *Boundary Components* is still possible. Under the assumption, that tasks are always component specific, the problem of input sequence generation is reduced to a search algorithm for non-distributed State Charts. To determine whether this assumption applies in general or if it applies only for systems with certain characteristics is part of the future work.

As mentioned above, there are multiple possible input sequences. They are chosen based on general or component specific test selection criteria. General criteria are:

Shortest Path: Only the shortest path between a components precondition and the components postcondition is traversed.

Path Coverage: Each possible path between a components precondition and the components postcondition is traversed.

Transition Coverage: Each possible transition between a components precondition and the components postcondition is at least traversed once.

Depending on the components' characteristics other selection criteria are thinkable. For example a GUI specific selection criterion would be input event coverage in combination with different input devices. The advantage of using component specific test selection criteria is, that the test case selection can be adapted to the specific properties of the component and therefore can better cover the corresponding error prone situations.

Based on the generation of the input sequences, the expected output of the test case can be extended by adding additional states, which must be fulfilled during the test. This applies for each state of a component, which is active on the path between the precondition and postcondition. As a consequence the verdict of a test case is more detailed, because the point during the execution of the test at which the system's behavior differs from the expected behavior can be located more precisely.

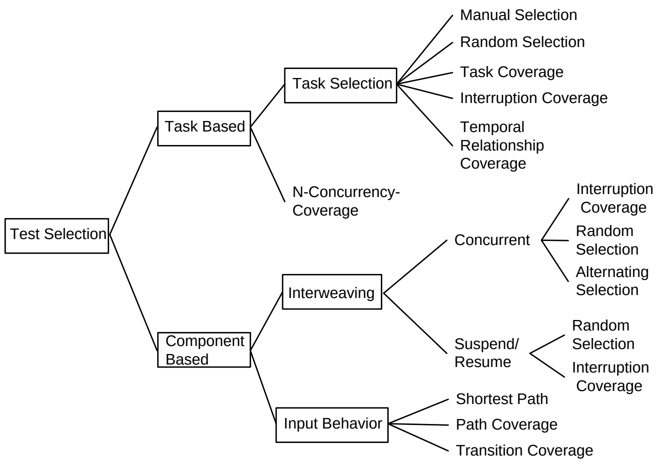


Figure 10: The different test selection criteria used during the coupled test case generation.

The generation of test cases from a task model is quite complex and produces a potentially large number of test cases. A complete coverage of all possible test cases is not realistic, due to the effort for the generation and execution of the tests. Figure 10 gives an overview of the proposed test selection criteria. The test selection criteria should be selected depending on the respective focus of the test. The focus depends on the system configuration and the used test setup. For example if the focus is to test whether the audio channel is correctly switched between different applications, the best criteria would be "interruption coverage" and "1-1-Concurreny-Coverage" on task level in combination with "Random Selection", "Interruption Coverage", and "Shortest Path" on component level. This combination covers all error prone interruption scenarios, but restricts the uncritical scenarios in this context to a minimum.

One of the resulting test cases for the second task sequence in Figure 8. The method calls are mapped by the test runtime environment to the corresponding driver components:

```
// task "Insert Disc"
checkState(CDPlayer.Disc_Unloaded)
triggerCdPlayer(ENV_Insert_Disc)
checkState(CDPlayer.Stopped)

// prepare GUI
triggerGui(ENV_Enter)

// task "Play"
checkState(GUI.Play_Button)
triggerGui(ENV_Enter)

// task "Move Forward"
checkState(SEAT.Idle)
triggerSeat(ENV_Forward)

// prepare Telephone
triggerTelephone(ENV_Network)

// task "Incoming Call"
checkState(Telephone.Network_Available)
triggerTelephone(ENV_Incoming_Connection)
triggerTelephone(ENV_Connection_Data_Received)
checkState(CDPlayer.Paused)
checkState(Telephone.Signal_Incoming_Call)
checkState(GUI.Accept_Button)

// task "Speak"
triggerGui(ENV_Enter)
checkState(TELEPHONE.Connected)
checkState(GUI.Active_Call)
triggerGui(ENV_Enter)
checkState(Telephone.Idle)
checkState(GUI.Play_Button)
checkState(CDPlayer.Playing)

triggerSeat(ENV_Stop)
checkState(SEAT.Idle)

// task "Eject"
triggerCdPlayer(ENV_Eject)
checkState(CDPlayer.Disc_Unloaded)
```

7. TOOL SUPPORT

The task models are create using the ConcurTaskTreeEnvironment (CTTE) [14]. These are imported in the test generation framework, which is realized using the Eclipse Modeling Framework (EMF) [4]. Using the EMF's meta modeling language ECORE the meta models for CTT and UML 2.0 State Chart Diagrams are defined. State Charts are currently the only supported method for specifying component models. Based on these meta models the relations between tasks and components are defined as shown in Figure 7. The task sequences are created by transforming the task model to a State Transition Network (STN) based on the in [13] proposed method. Currently the only supported selection criteria are manual task selection and task coverage. The generation of executable test cases based on the task sequences is performed using a shortest path algorithm for state charts. Target for the test cases generation is the test case specification language of a test automation framework developed by BMW Group. This framework supports specific driver components for the automotive domain. The tool is integrated into the BMW development process by providing import functionalities for existing specifications (e.g. Bus Interface specificiations defined in Rhapsody or the UI specification, defined in a domain specific language).

8. RELATED WORK

There are several approaches for test case generation from use case models. Use case models describe a system at the same abstraction level as task models, but they lack a formal foundation for the relationships between the use cases. In [7] Use Cases are mapped onto State Charts using preconditions and postcondition. Based on this relations test cases for each use case are semi-automatically generated from the State Charts using AI planning. The critical interaction between different use cases are not regarded in the test case generation.

The work presented in [21] is based on the formalization of user requirements in services. Each service is linked to functional models on lower abstraction levels. Based on these relations service specific test cases are generated. Likewise the relations between different services are not included in the test case generation.

In [5] test cases for integration testing are generated based on Use Case Models. The Use Cases are extended by interruption points. Based on these a CSP behavior model is generated, which is used for the test case generation. The approach is limited to interruptions between different features and considers no other feature relations like disabling, concurrency or enabling.

9. CONCLUSION

The approach presented in this paper enables the generation of test cases for integration testing that cover the typical error prone interactions between different system components. This is accomplished by modeling the tasks the system can perform in interaction with its environment.

The tasks and their relations are described using the ConcurTaskTree (CTT) notation. The advantage of CTTs is the formal definition of the temporal relationships between tasks and the easy and understandable notation. Based on a CTT test cases are generated in form of task sequence using task specific test selection criteria. These selection criteria allow the explicit selection of critical interaction scenarios, which are not covered by the typical integration models at the communication level. By instantiating the task sequences using the component models, it is possible to create detailed test cases, which can be directly executed. The instantiation is possible due to an explicit mapping between test model and component models. The missing input behavior and expected output behavior of a task sequence can be derived from the component models using this mapping. Therefore the generated tests have the same abstraction level as the component models and no integration test specific driver components have to be created.

The presented approach for test case generation has been successfully applied to an existing infotainment system specification of BMW Group. The next steps are a further formalization of the described method and a comprehensive evaluation of the generated test cases by testing the implementation of the specified system.

10. REFERENCES

[1] Y. Ait-Ameur and M. Baron. Formal and experimental validation approaches in HCI systems design based on a shared event B model. *Int. J. Softw. Tools Technol. Transf.*, 8(6):547–563, 2006.

[2] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Comput. Netw. ISDN Syst.*, 14(1):25–59, 1987.

[3] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner. *Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[4] F. Budinsky, S. A. Brodsky, and E. Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[5] A. L. L. de Figueiredo, W. L. Andrade, and P. D. L. Machado. Generating interaction test cases for mobile phone systems from use case specifications. *SIGSOFT Softw. Eng. Notes*, 31(6):1–10, 2006.

[6] R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2, 1971.

[7] P. Fröhlich and J. Link. Automated test case generation from dynamic models. In *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 472–492, London, UK, 2000. Springer-Verlag.

[8] M. Gupta, F. Bastani, L. Khan, and I.-L. Yen. Automated test data generation using MEA-Graph planning. In *ICTAI '04: Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'04)*, pages 174–182, Washington, DC, USA, 2004. IEEE Computer Society.

[9] H. R. Hartson, A. C. Siochi, and D. Hix. The UAN: a user-oriented representation for direct manipulation interface designs. *ACM Trans. Inf. Syst.*, 8(3):181–203, 1990.

[10] A. E. Howe, A. von Mayrhauser, and R. T. Mraz. Test case generation as an AI planning problem. *Automated Software Engg.*, 4(1):77–106, 1997.

[11] B. E. John and D. E. Kieras. The GOMS family of user interface analysis techniques: comparison and contrast. *ACM Trans. Comput.-Hum. Interact.*, 3(4):320–351, 1996.

[12] P. Johnson, H. Johnson, R. Waddington, and A. Shouls. Task-related knowledge structures: analysis, modelling and application. In *Proceedings of the Fourth Conference of the British Computer Society on People and computers IV*, pages 35–62, New York, NY, USA, 1988. Cambridge University Press.

[13] K. Luyten, T. Clerckx, K. Coninx, and J. Vanderdonckt. Derivation of a dialog model from a task model by activity chain extraction. In *DSV-IS*, pages 203–217, 2003.

[14] G. Mori, F. Paterno, and C. Santoro. CTTE: Support for developing and analyzing task models for interactive system design. *IEEE Transactions on Software Engineering*, 28(8):797–813, 2002.

[15] L. Nóbrega, N. J. Nunes, and H. Coelho. Mapping ConcurTaskTrees into UML 2.0. In S. W. Gilroy and M. D. Harrison, editors, *DSV-IS*, volume 3941 of *Lecture Notes in Computer Science*, pages 237–248. Springer, 2005.

[16] Object Management Group. *OMG Unified Modeling Language Specification*, März 2003.

[17] A. J. Offutt, Y. Xiong, and S. Liu. Criteria for generating specification-based tests. In *ICECCS '99: Proceedings of the 5th International Conference on Engineering of Complex Computer Systems*, page 119, Washington, DC, USA, 1999. IEEE Computer Society.

[18] F. Paternò. *Model-Based Design and Evaluation of Interactive Applications*. Springer-Verlag, London, UK, 1999.

[19] F. Paternò, C. Mancini, and S. Meniconi. ConcurTaskTrees: A diagrammatic notation for specifying task models. In *INTERACT*, pages 362–369, 1997.

[20] F. Paternò and C. Santoro. Integrating model checking and HCI tools to help designers verify user interface properties. In *DSV-IS*, pages 135–150, 2000.

[21] C. Pfaller, A. Fleischmann, J. Hartmann, M. Rappl, S. Rittmann, and D. Wild. On the Integration of Design and Test - A Model Based Approach for Embedded Systems. *Proceedings of the Workshop on Automation of Software Test (AST 06)*, 2006.

[22] W. Prenninger, M. El-Ramly, and M. Horstmann. Case studies. In *Model-Based Testing of Reactive Systems*, pages 439–461, 2004.

[23] W. Prenninger and A. Pretschner. Abstractions for model-based testing. *Electr. Notes Theor. Comput. Sci.*, 116:59–71, 2005.

[24] S. Reid. The art of software testing, second edition. glenford j. myers. revised and updated by tom badgett and todd m. thomas, with corey sandler. john wiley and sons, new jersey, u.s.a., 2004. isbn: 0-471-46912-2, pp 234: Book reviews. *Softw. Test. Verif. Reliab.*, 15(2):136–137, 2005.

[25] R. E. K. Stirewalt. MDL: A language for binding user-interface models. In *CADUI*, pages 159–170, 1999.

[26] G. van der Veer, B. Lenting, and B. Bergevoet. GTA: Groupware task analysis - modeling complexity. *Acta Psychologica*, 1996.

[27] J. A. Whittaker. Stochastic software testing. *Ann. Softw. Eng.*, 4:115–131, 1997.