Synthesizing Tests for Combinatorial Coverage of Modal Scenario Specifications

Valerio Panzica La Manna
MIT Media Lab
Cambridge, MA, USA
Email: vpanzica@mit.edu

Itai Segall
Bell Labs Israel
Alcatel-Lucent
Email: itai.segall@alcatel-lucent.com

Joel Greenyer
Software Engineering Group,
Leibniz Universität Hannover
Welfengarten 1 30167 Hannover, Germany
Email: greenyer@inf.uni-hannover.de

Abstract—**Software-intensive systems often consist of many components that interact to fulfill complex functionality. Testing these systems is vital, preferably by a minimal set of tests that covers all relevant cases. The behavior is typically specified by scenarios that describe what the system may, must, or must not do. When designing tests, as in the design of the system itself, the challenge is to consider interactions of scenarios. When doing this manually, critical interactions are easily overlooked. Inspired by Combinatorial Test Design, which exploits that bugs are typically found by regarding the interaction of a small set of parameters, we propose a new test coverage criterion based on scenario interactions. Furthermore, we present a novel technique for automatically synthesizing from Modal Sequence Diagram specifications a minimal set of tests that ensures a maximal coverage of possible t-wise scenario interactions. The technique is evaluated on an example specification from an industrial project.**

I. Introduction

In many application domains, from embedded systems to distributed information systems, we find systems that consist of multiple interacting software components that must fulfill complex functionality in reaction to external events. To elicit and specify the system behavior, most design methodologies are based on use cases and scenarios in order to describe how the system may, must, or must not react in certain situations.

From a scenario-based specification, a system must be implemented that satisfies all the previously specified scenarios. This is especially difficult if multiple scenarios can occur at the same time and then *interact*, i.e., imply restrictions upon each other. This can happen for example due to concurrent user requests or concurrent processes in the physical environment. When designing *tests*, it is therefore crucial to cover especially cases where scenarios interact. At the same time, there should not be too many tests in order to avoid an unnecessary cost and time overhead, especially if tests are executed manually.

To address this problem, we propose a new technique to synthesize a test suite, i.e., a set of tests, from a scenario-based system specification. Novel in our technique is, first, our definition of a scenario-based coverage criterion as a measure of quality of the produced tests. The criterion is motivated by Combinatorial Test Design (CTD) [24], which exploits that a software bug is typically found by forcing, in the tests, the interaction of a small set of parameters [5], [23], [15]. We transfer this idea to scenario-based specifications and define a measure of how many of the *possible pairwise (or t-wise) concurrent activations of possibly interacting scenarios* are forced by a test suite. Second, we present a new game-based approach for synthesizing a test suite from a scenario specification. The resulting test suite guarantees reaching a full pairwise (or t-wise) test coverage while at the same time ensuring a *minimal* number of tests.

We propose to model scenarios using Modal Sequence Diagrams (MSDs), a formal interpretation of UML sequence diagrams [12] based on the concepts of Live Sequence Charts (LSCs) [6], [13]. MSDs allow engineers to visually specify interaction behavior that may, must, or must not happen among system and environment components. This formalism is particularly suited for our purpose for two reasons. First, it allows engineers to intuitively, yet formally, design scenarios that specify safety properties (“something bad must not happen”) as well as liveness properties (“something good must eventually happen”). Second, the MSDs semantics defines the independent activation and progress of scenarios, which naturally reflects how multiple use cases can occur concurrently.

Test suites are synthesized from these specifications intuitively as follows. We view the interaction of the system and its environment as a two-player game. We explore if the environment, i.e., eventually the tester or test software, will be able to interact with the system in such a way that concurrent activations of MSDs will take place where the active MSDs share at least one common object. It may be that an MSD specification contains non-determinism, and for example allows the reactions of the system to occur in different orders or allows it to take different actions. Depending on the choice made by the system (usually the non-determinism is resolved in the final implementation), events in the environment, in turn, may have to be selected differently to force a certain concurrent activation of MSDs. Our technique is able to deal with this non-determinism. As a result, a test synthesized by our technique is not only a sequence of environment and system events—instead a test is a *strategy* that may at some points provide for different possible system reactions and, in turn, prescribes different subsequent environment events.

The approach was implemented as part of ScenarioTools, an Eclipse-based tool suite for the modeling and analysis of MSD specifications [1], [11]. We evaluated our approach on a specification taken from an industrial system. Since the system is not yet implemented, we automatically synthesized an implementation from the specification for evaluation. We use a simplified specification as a running example.

The paper is structured as follows. In Sect. II we describe the preliminaries of MSDs and CTD. We then introduce our coverage criterion in Sect. III. In Sect. IV we present our test suite synthesis approach, and in Sect. V its evaluation. In Sect. VI we present the related work and conclude in Sect. VII.

II. Background

In the following, we describe our running example and the foundations of MSDs and CTD.

A. Example

We consider a server repair and maintenance system that manages the runtime (or *online*) replacement and insertion of nodes into a server. Especially, we consider the insertion of a node, which is also called *expansion* of the server. For this purpose, the maintenance system interacts with a system administrator, in the following simply called the *user*, via a *console*. Furthermore, the maintenance system has a verification unit that must ensure that certain hardware components are working correctly.

The online expansion operation consists of three main parts: a pre-operation verification, the expansion operation, and post-operation verification. In the pre-operation verification, the system checks the status of its internal and external components. In our example, three components are checked: the network, the existing hardware, and the LED lights (which will guide the user to the correct slot in which the node should be inserted). At this stage, a failure in the network or the hardware is considered a non-recoverable error, i.e., an error which causes the operation to abort. A broken LED, on the other hand, is a recoverable error, which is reported to the user. The user is notified of recoverable errors and can decide to abort or continue the operation.

After a successful pre-operation verification, the user is instructed to physically insert the hardware node. Once the node is inserted, another round of verification is performed. The post-operation verification is similar to the pre-operation verification. The difference here is only that a post-operation network failure is considered a recoverable error. A working network is essential to perform the expansion, but the inserted node will work if a post-operation network error occurs and can be fixed.

The example highlights the following issues. First, there can be different overlappings of the scenarios. For example, different combinations of failures can occur at the same time and in different orders. Second, we consider a case in the example where the specification contains a non-deterministic choice. For brevity of the example, we consider a very simple case in which the system shows the user either a continue dialog or an OK/Cancel dialog in the event of a recoverable error. Depending on the dialog implemented in the final system, the user will have to react differently in order to continue the test. In bigger specifications, the effect of the non-determinism on the different subsequent environment events can be more complicated.

Third, our example contains the case where the same message (the network error being detected) causes a different behavior when appearing at different points in the process. This is often overlooked in manual test design that only considers statically whether some events occur or not. For example, a test architect may think of the case where the network fails, but may fail to see that the same network failure at different points in time should cause different behaviors, and thus should be tested separately.

B. MSD Specifications

MSDs are a formal interpretation of UML sequence diagrams introduced by Harel and Maoz [12]. An MSD specification consists of a set of MSDs that specify the valid interactions of components in a system. In this work, we focus on *universal* MSDs, which describe properties that must be satisfied by every execution of the system. Our technique can be extended to also support *existential* MSDs.

In the following, we call the components more generally *objects*. We consider open systems in which objects are either *controllable* or *uncontrollable*. Controllable objects are the (software) components under development; uncontrollable objects are users or external components. In the following, we call the set of uncontrollable objects *environment*; the set of controllable ones we call the *system*.

Figure 1 shows part of the MSD specification for our server repair and maintenance system. The object system is modeled by a composite structure diagram (CSD) shown on the top left. The communicating objects are represented by roles that have a cloud-like shape if the object is uncontrollable and a rectangular shape if the object is controllable.

The objects communicate by interchanging messages that have a name and one sending and one receiving object. For simplicity, we consider only synchronous messages where the sending together with the receiving of a message is one event, also called a *message event*. Our approach can, however, be extended to also support asynchronous communication. Messages sent from uncontrollable (resp. controllable) objects are also *uncontrollable events* (resp. *controllable events*).

Like a role in the CSD, each lifeline in an MSD represents an object. Messages in an MSD represent message events. In universal MSDs, messages can have a *temperature* and an *execution kind*. The temperature can be either *cold* or *hot*; the execution kind can be either *monitored* or *executed*. Hot message are red and cold messages are blue; Monitored messages have a dashed arrow, and executed messages have a solid arrow. For clarity, we also use the labels (h/c, m/e).

The temperature and execution kind encode safety resp. liveness properties of events in a scenario. Intuitively, if a scenario reaches an executed message, the corresponding message must eventually occur. If the message is monitored, this message may occur. Hot messages must not be *violated*, which happens if a message event occurs that is expected only at another point in the scenario. Cold messages can be violated, and this violation will terminate the scenario.

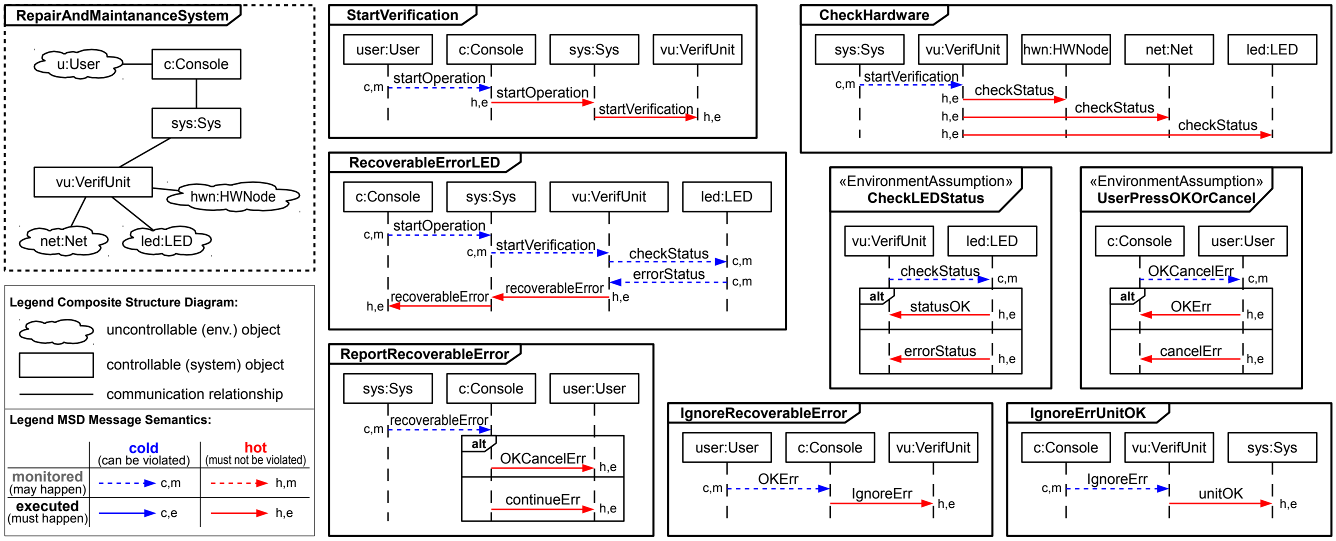

Fig. 1. Part of the MSD specification of the runtime server management system

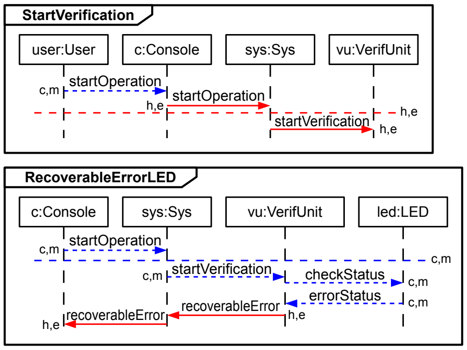

Fig. 2. Concurrently active MSDs with different temperature and execution kind of their cuts

As an example, consider the MSD StartVerification shown in Fig. 1. The first message in an MSD is always cold and monitored. Here it says that the scenario starts when the user tells the console to start the (expansion) operation. The next message is executed, which means that next the console must forward this request to the maintenance system. Likewise, the system must subsequently order the verification unit to start the verification. The hot temperature of the second message means that the console must send the start operation message to the system before the user starts another expansion operation and before the system orders the start of the operation.

More precisely, the semantics of the message in a universal MSD is as follows. A message event can be *unified* with a message in the MSD if the event name equals the message name and the sending and receiving objects of the message event are represented by the sending and receiving lifelines of the diagram message. When an event occurs that can be unified with the first message of an MSD, an *active copy* of that MSD is created, also called *active MSD*. We assume that an MSD has only one first message. As further events occur that can be unified with the subsequent messages in the MSD, the active MSD progresses. This progress is captured by the *cut*, which marks for every lifeline the occurred messages. If the cut reaches the end of an active MSD, it terminates.

If the cut is in front of a message on its sending and receiving lifelines, this message is *enabled*. If a hot message is enabled, the cut is *hot*, otherwise the cut is *cold*. Similarly, if an executed message is enabled, the cut is *executed*, otherwise the cut is *monitored*. Figure 2 shows active copies of the MSDs StartVerification and RecoverableErrorLED that are in a hot and executed cut, resp. in a cold and monitored cut. The cuts are represented by dashed horizontal lines in the diagram.

As shown in Fig. 2 there can be multiple active MSDs at a time, which impose different requirements on what must or must not occur next. The cut configuration illustrated in Fig. 2 occurs after the occurrence of the message events `user->c:startOperation` and `c->sys:startOperation`.

Continuing the explanation of the example, the MSD RecoverableErrorLED says that if after starting the verification and checking the status of the LED component, this component returns an error status, this must be reported as a recoverable error to the user. The checking of the LED, the network, and the server hardware must follow the start verification request as described by MSD CheckHardware. The MSD ReportRecoverableError says that the error must be reported to the user either by an OK/Cancel dialog or a continue dialog. If, in case an OK/Cancel dialog is implemented by the system, the user presses OK, the verification unit is told to ignore the error (MSD IgnoreRecoverableError) and then tell the system that the verification of the component was OK (IgnoreErrUnitOK). The further MSDs of the example are omitted for brevity.

One general assumption, to keep the environment from trivially violating the MSD specification, for example by sending `user->c:startOperation` twice without allowing the system to send `c->sys:startOperation` etc., is that the system is always fast enough to execute any finite number of steps before the next environment event occurs. This assumption is also called the *synchrony hypothesis* or *synchrony assumption* [12], [13].

In some applications, even further assumptions have to be made about the environment of the system, otherwise it may even be that no implementation for the specification exists. In previous work, we proposed to model these assumptions explicitly by also using MSD, which we then call *assumption MSDs* [10]. Figure 1 shows for example the assumption MSD RecoverableErrorLED that describes the assumption that the LED component, if requested to check the status of the LEDs, will eventually reply with an OK status or an error status. Similarly, the assumption UserPressOKOrCancel describes the assumption that the user, if presented with an OK/Cancel dialog for an error, will eventually choose OK or Cancel.

C. Game Graphs

Our test synthesis technique relies on exploring different sequences of uncontrollable and controllable events that lead to different combinations of active MSDs. We do this by building a *game graph*, which is essentially a labeled transition system where transitions are labeled with message events and where states represent different reachable configurations of active MSDs and their cuts. As events are controllable and uncontrollable, we also call the respectively labeled transitions *controllable* and *uncontrollable*. The initial state corresponds to a state where no copy of any active MSD was created.

Definition 1 (Game Graph): A game graph is a tuple $G = (Q, \Sigma, T, q_0)$ where Q is a finite set of states and q_0 the initial state. Σ is a finite set of message events. $T \subseteq Q \times \Sigma \times Q$ is a transition relation. Σ_u and Σ_c with $\Sigma = \Sigma_u \cup \Sigma_c$ and $\Sigma_u \cap \Sigma_c = \emptyset$ are the uncontrollable resp. controllable message events. $T_u = T \backslash Q \times \Sigma_c \times Q$ are the uncontrollable transitions; $T_c = T \setminus T_u$ are the controllable transitions.

In Sect. IV, we describe an algorithm that explores the game graph of an MSD specification, called *specification game graph* in the following, in order to find test strategies. Test strategies are also represented as game graphs and they can be extracted from the specification game graph.

Due to the synchrony hypothesis mentioned earlier, we will regard only game graphs where states have either controllable or uncontrollable outgoing transitions. We call a state a *controllable state* or *system state* if it has only outgoing controllable transitions and we call a state a *uncontrollable state* or *environment state* if it has only outgoing uncontrollable transitions.

D. Combinatorial Test Design

Combinatorial Test Design (CTD), a.k.a. combinatorial testing, is an effective test planning technique aimed at exercising interactions between parameters. In CTD, the test space is manually modeled by a set of parameters, their respective values, and restrictions on the value combinations, a.k.a. a combinatorial model. A valid test in the test space is defined to be an assignment of one value to each parameter without violating restrictions. A subset of the space is automatically constructed so that it covers all valid value combinations (a.k.a. interactions) of every t parameters, where t is usually a user input. The most common application of CTD is known as pairwise testing, in which the interaction of every pair of parameters must be covered (i.e. t=2). Each test in the result of CTD represents a high level test, or a test scenario, that needs to be translated to a concrete executable test.

The reasoning behind CTD is the observation that in most cases, the appearance of a bug depends on the interaction between a small number of features, or parameters, of the system under test. Experiments show that a test set that covers all possible pairs of parameter values can typically detect 50% to 75% of the bugs in a program [5], [23]. Other experimental work has shown that typically 100% of bugs can be revealed by covering the interaction of between 4 to 6 parameters [15].

III. MSD-based Testing

Traditional testing techniques require the test architect to manually design a test plan on the basis of requirements and design documents that informally describe the functionality of the system under test. The lack of a formal specification may result in an ineffective or incomplete test suite.

In this section we describe how MSD-based testing simplifies the process of designing a test plan. We then provide the definition of test strategies, the result of our synthesis technique. Finally, we introduce the coverage criterion based on the MSD specification as a measure of the quality of the produced test suite.

A. Specifying Goal States

The process of designing a test plan traditionally requires the engineer to manually generate, from an informal specification, relevant test cases for specific functionality of the system. The manual generation of tests requires the identification of specific sequences of events that can occur in the environment and the corresponding system reaction. This process is challenging for conventional desktop applications and can be very difficult for complex reactive systems.

When designing a test plan, it is important to decide when a test should end. One can argue that a test ends if all scenarios are terminated. However, there may be systems where some scenarios never terminate; our production cell example, mentioned in Sect. V, is such an example. In our industrial example, we discovered that test engineers typically want to define specific end-conditions for tests.

Therefore, we require a way to define appropriate end-points for tests. End-points can be characterized in many ways, including certain diagrams being active, specific cuts in certain diagrams achieved, a parameter being set to a specific value, etc. We call states of a system that satisfy this end-point condition *goal states*.

We choose to adopt the former, and introduce *goal diagrams*. Given a set of diagrams marked as goal diagrams, a state is a goal state if a goal diagram is in the last cut. Testers

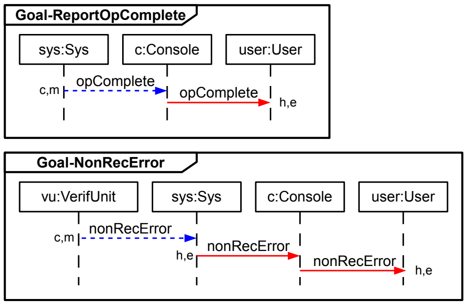

Fig. 3. Goal Diagrams of the example

can select certain MSDs in the specification as goal diagrams or add new MSDs as goal diagrams that, when reaching the end, mark a desired end point of a test.

Figure 3 shows the goal diagrams of our running example. The diagram Goal-ReportOpComplete describes the end-point of the successful completion of a runtime maintenance operation and Goal-NonRecError represents the scenario in which the system notifies the user of a non-recoverable error.

There may be cases where test engineers do not specify any specific end points for a test. Also in these cases we need to know when a test should end, and hence we also need a generic notion of goal states. Since our test strategy synthesis is based on the specification game graph, tests should not unnecessarily traverse cycles in this graph. From the perspective of the specification, this would not cover any new behavior. Therefore, we define goal states generically, if no goal diagrams are defined, as the leaf nodes of a spanning tree of the specification game graph. This spanning tree can be determined by a depth- or breadth-first-search.

B. *Test Strategy*

MSD-based specifications, as also other specifications, are typically *under-specified*, which means that they do not deterministically specify the system's behavior in every possible situation. In MSD specifications, non-determinism often occurs because concurrently active MSDs can progress in different orders. A specification-based test generation technique must therefore take this non-determinism into account.

This gives rise to the notion of test strategies. The non-determinism may be resolved differently in the final implementation. Therefore, the tests generated from the specification are not only sequences of environment and system events. Instead a test is a strategy that, after some sequence of events, may provide for different reactions of a system that again require a specific subsequent environment event. A test strategy is a game graph in which the tester (or the test engine) plays the role of the environment against any system implementing the MSD specification. For our purposes, we require strategies to only prescribe deterministic choices for the environment. This means that a tester that plays the role of environment during a test should never have to make a non-deterministic choice. Finally, test strategies should ensure reaching a goal state.

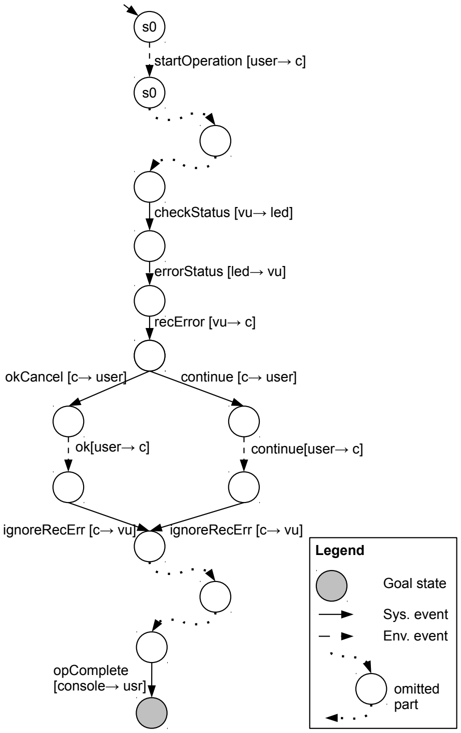

Fig. 4. Test Strategy

Definition 2 (Test Strategy): A test strategy for a specification S is a game graph in which each state has at most one outgoing transition labeled with an environment event, and that for each correct system implementing S, reaches a goal state.

Figure 4 shows one test strategy[1] generated by our technique from the MSD specification of the example. The test strategy has one goal state associated to the goal diagram Goal-ReportOpComplete, which describes the scenario in which the user is notified of the successful completion of the maintenance operation. The example shown in the figure contains only one goal state. However, in general, a test strategy may contain multiple goal states.

Transitions drawn as dashed lines represent instructions for the tester while the solid lines represent moves of the system. States with outgoing transitions labeled with environment events indicate a turn of the tester, i.e., a point in which the tester has to send the associated environment event to the system. States with outgoing transitions labeled with system events represent a turn of the system. Since the specification is non-deterministic it is allowed for these states to have multiple outgoing transitions. A test strategy will drive the test to its goals for *every* possible system that satisfies the specification.

C. *Coverage Criterion*

We now propose an MSD-based coverage criterion for test strategies. As with most coverage criteria, the criterion may be used for two purposes: a) given a test suite, minimize it

[1] due to space limit, we omitted some parts of the test strategy and show only the ones that help illustrating Def. 2.

by selecting a small subset that maintains the same overall coverage as the given set, and b) for evaluating the expected effectiveness of a given test suite, or a given set of test executions. In this work we focus on the former.

Given an MSD specification of a system, we propose a coverage criterion for a set of test strategies which is based on coverage of its diagrams and their interactions. We adopt the motivation behind CTD [5], [23], [15], which states that defects in a system are commonly caused by the interaction of a small number of elements. In our case, the elements to reason about are active MSDs. We therefore propose the *t-wise active diagrams* coverage criterion. Intuitively, in this criterion, every tuple of up to t diagrams should be active concurrently during the run, if at all possible, where t is a user-specified number.

In the following we formalize the definition of this coverage criterion. Intuitively, we start by defining a tuple of diagrams as *valid* if one can reach a state in which all diagrams in it are active. We then define that a test strategy *covers* a tuple of diagrams if a state in which all diagrams in the tuple are active will be visited by this test strategy. Finally, we define the coverage achieved by a set of test strategies as the number of covered diagram tuples (up to a given size t), out of all valid diagram tuples (up to the same size t). One achieves full t-wise coverage by introducing a set of test strategies that forces every valid set of diagrams up to a size of t to be covered, i.e., forces the system to reach a state in which they are all active concurrently.

Definition 3 (Valid Diagram Tuple): A tuple of diagrams, D, is called *valid* if there exists a path in the specification game graph that reaches a state in which all diagrams in D are active.

Definition 4 (Coverage of a Diagram Tuple): A diagram tuple D is *covered by a test strategy* if every path in the strategy from the initial state to a goal state contains at least one state in which all diagrams in D are active. Furthermore, we say that a set S of diagram tuples is covered by a set of test strategies if for every D in S there exists a test strategy in the set that covers D.

Definition 5 (t-wise Active Diagrams Coverage): Given a set of test strategies, their t-wise active diagrams coverage is the number of tuples of size up to t that are covered by the set of strategies divided by the set of all valid tuples of size up to t.

For example, consider the diagrams depicted in Figure 1, and described in Section II-B above. Any test strategy for this system that starts with the message `startOperation` guarantees coverage of the pair of diagrams given in Figure 2 since both are activated by the `startOperation` message. The diagram Goal-ReportOpComplete in Figure 3 (and similarly any pair of diagrams containing it) is not covered by a test strategy in which an error is detected in the pre-op verification stage since the `opComplete` message is never reached in such cases. The pair of diagrams in Figure 3 is an invalid pair—in no strategy can a non-recoverable error happen concurrently with the successful completion of the maintenance operation.

IV. Approach

In this section we describe our approach for synthesizing a minimal set of test strategies from an MSD specification that has maximal t-wise Active Coverage criterion for a given t.

The approach consists of the following steps: (i) given as input the MSD specification and a set of goal diagrams, we explore the game graph of the MSD specification and as a result synthesize a *Global Test Strategy Graph* (GTSG), which contains all tests strategies; (ii) we leverage the information of active MSDs associated to states of the GTSG to compute for each strategy contained in the GTSG the set of covered diagram tuples; (iii) we compute a minimized set of test strategies contained in the GTSG that satisfies the t-wise coverage criterion by applying a well-known greedy test minimization algorithm; (iv) for each test contained in the minimized set, we finally extract from the GTSG the concrete, separate test strategies. From these test strategies, (v) for example test plan documents or code for automated tests can be generated, or the test strategies can be simulated to guide the tester through tests interactively. These features, however, are not the focus of this paper.

A. Global Test Strategy Graph Synthesis

In order to find a minimal set of test strategies for an MSD specification with maximal t-wise active diagram coverage, we must first compute all test strategies. Furthermore, we must compute all valid tuples, in order to determine the maximal coverage guaranteed by all test strategies, which is also the coverage that should still be guaranteed by the minimized set of test strategies. A naïve approach would compute the set of all valid tuples while extracting each test strategy separately. However, *all* the valid tuples and *all* the test strategies can be discovered by a *single* exploration of the specification game graph. In the following, we explain our algorithm for this exploration. Given a set of goal diagrams, the algorithm synthesizes a game graph, called the *Global Test Strategy Graph* (GTSG), which contains all the valid tuples and all the test strategies reaching the given goals.

Our exploration and synthesis algorithm is based on the on-the-fly algorithm for solving reachability games proposed by Cassez et al. [4] (see also David et al. [7]). The algorithm was also adopted by Greenyer et al. for synthesizing controllers from MSD specifications [11].

The problem of synthesizing the GTSG can be viewed as the problem of finding winning strategies in a two-player reachability game, played by the environment against the system. In our context, the tester takes the role of the environment. We assume that the tester can control all the events that are uncontrollable by the system. The environment wins the game if the environment can guarantee, by selecting uncontrollable events in the right way, to reach a goal state, while the system tries everything (by selecting controllable events) to prevent the environment doing so. If the environment wins, it means that there exists at least one test strategy. The algorithm then continues the exploration of the state space until it finds all the possible test strategies.

The algorithm performs a depth-first, forward exploration of the game graph. When it finds a goal state, this state is marked as *winning*. A state is identified as a goal state if in that state, one of the given goal diagrams is in a terminal cut. If no goal diagrams are specified (cf. Sect. III-A), goal states are states that have no unvisited successors.

Whenever the algorithm finds a goal or winning state, it performs a backward re-evaluation of the winning status of predecessor states. In our context, a state is winning if either it is a goal state, or all outgoing controllable (i.e., system-controllable) transitions from it lead to winning states, or at least one outgoing uncontrollable (i.e., environment-controllable) transition from it leads to a winning state. The backward re-evaluation continues along the chains of predecessors until the initial state has also been re-evaluated. If the initial state is a winning state, test strategies can be extracted from the specification state graph. The set of winning states induces the GTSG, i.e., the GTSG can be extracted from the explored specification game graph by removing from it non-winning states and also removing the transitions to/from non-winning states.

B. Collecting Strategies and their Coverage

A GTSG encompasses test strategies that differ in the environment choices taken from each state in the graph, and they also differ in their coverage. We therefore require to identify the different test strategies in a GTSG, and the coverage guaranteed by each. Explicit extraction of each of the test strategies, however, entails a large memory overhead. Therefore, we symbolically annotate in the GTSG the different strategies contained in it and calculate and compare their prospective coverage.

This is performed in a backward propagation manner: starting from the goal states and moving backward towards the root, we identify for each state the different sub-strategies for the environment to reach a goal state from this state. In a state there are multiple sub-strategies if it has multiple outgoing transition labeled with an environment event.

In this backward propagation, we label a state, *per* sub-strategy that starts from that state, with a set of valid tuples of active MSDs that this sub-strategy can guarantee to cover. Thus, a state is annotated by a set of set of tuples—which set of tuples corresponds to which sub-strategy is implied by the GTSG and the (previously created) labels of the successor states.

At the end of this process, the root will be labeled with several sets of tuples. Each set represents the tuples that are guaranteed to be covered by one test strategy. The single test strategies can be identified by these tuples, but later have to be extracted into an explicit strategy description. This requires another forward exploration in the GTSC, which will be described in Sect. IV-D. However, since a set of tuples that is contained in the label of a state represents a test (for the initial state of the GTSC) or a partial test (for the other non-initial states of the GTSC), we refer to such a tuple as an *abstract test*.

Definition 6 (Abstract Test): An abstract test in a state s is the set of diagram tuples for which a strategy exists that guarantees covering the diagram tuple starting from s.

For a given state s, we denote by A_s the set of all abstract tests in it. We now present an algorithm for computing A_s for all states in the test strategy. The algorithm proceeds from the goal states backward towards the initial state, and applies in each state the following rules.

Given a state s, we denote by $AT(s)$ all the diagram tuples (up to size t) that are active in state s, i.e., the coverage targets covered in state s. The set A_s is: (i) If s is a goal state, then A_s contains exactly one element, consisting of the set $AT(s)$. (ii) If s is an environment state (a state from which outgoing edges are environment events), then let $s_1, \ldots, s_k$ be the states reachable by a single transition from s, and $A_{s_1}, \ldots, A_{s_k}$ be the respective sets of abstract tests. $A_s = \{AT(s) \cup t | \exists i.t \in A_{s_i}\}$. In words, for each abstract test in a "child" of s, there exists an abstract test in s consisting of the tuples in the abstract test of the child, as well as all the tuples active in s. (iii) If s is a system state (a state from which outgoing edges are system events), then let $s_1, \ldots, s_k$ be the states reachable by a single transition from s, and $A_{s_1}, \ldots, A_{s_k}$ be the respective sets of abstract tests. $A_s = \{AT(s) \cup (t_1 \cap t_2 \cap \cdots \cap t_k) | t_1 \in A_{s_1} \wedge t_2 \in A_{s_2} \wedge \cdots \wedge t_k \in A_{s_k}\}$. In words, a test strategy from s would consist of a test strategy from each of its children (because we don't know in advance what the system will choose to do from s). Therefore, we take the tuples common to all of its children (those are the ones that can be guaranteed for every system), and then add the tuples covered "locally" in s.

At the end of the process, each abstract test in the initial state captures exactly the set of diagram tuples (up to size t) that a tester can guarantee to cover in a single test strategy. Note that this coverage is guaranteed for any system implementing the specification.

This algorithm constructs abstract tests for all possible test strategies from each state. It is often the case that two test strategies are equivalent in the sense that they guarantee the exact same coverage (i.e., their abstract tests are equal). In such cases, one can remove the duplicate abstract tests and retain only one copy of them. This way, equivalent test strategies (in terms of their guaranteed coverage) are identified early in the process, and significant performance improvement is gained.

C. Minimization

In test suite minimization, one minimizes a test suite to a subset of tests that maintains the same coverage for a certain coverage criterion. We adopt this notion, and minimize our test suite (which is in fact a suite of test strategies) based on the t-wise active diagram coverage criterion.

Test suite minimization algorithms [25] typically consider as an input a table, in which a row corresponds to a test, and columns correspond to coverage targets. A value 1 in a cell represents that this test covers this target, and 0 that it does not. In our case, each column represents a valid diagram tuple of size up to t, and each row represents a test strategy. A value of 1 in the table captures that the strategy guarantees coverage of the target. Existing test suite minimization algorithms [25] can then be applied on this table, in order to minimize it and choose a small subset of strategies without reducing the guaranteed coverage.

One should note that 100% coverage is not guaranteed, even without minimization. This is due to the previously discussed non-determinism in the specification. Often this results in cases where a certain tuple of diagrams is valid (i.e., there exists a system and environment such that this tuple would be visited), but not necessarily guaranteed for *every* system.

D. Test Strategy Extraction

As described in Section IV-B above, test strategies are not explicitly collected in the traversal algorithm, but rather only abstract representations thereof, in the shape of their guaranteed coverage. However, when one comes to display and use (e.g. execute) the test generation and minimization results, the concrete test strategy is required.

We therefore also supply a method for extracting a concrete test strategy, given an abstract test. The method traverses the test strategy once again, top-down (i.e. from root to goals). During the traversal, a list of required diagram tuples is maintained. This list is initialized with the complete required abstract test. In each state, the tuples covered by this state are removed from the list, and traversal continues to its children. For system states, all children are included in the extracted strategy. For environment states, one child is chosen such that it contains an abstract test that contains the required list.

Last, the concrete test strategies can be compiled into executable test programs. We did not implement such a translation. Rather, for our evaluation, we implemented an interpreter within SCENARIOTOOLS to execute the test strategies.

V. EVALUATION

To evaluate our approach, we considered three comprehensive specifications of our server maintenance system example. Each specification represents a separate supported operation. The specifications consist of between 73 and 78 MSDs, out of which 4 are goal diagrams: capturing a successful completion, a recoverable error, a non-recoverable error, and a critical error.

While the system has not been implemented, the MSD specification is still derived from the existing English requirements document of the real industrial system to be implemented.

For each specification, a test controller was generated, containing around 400 candidate test strategies. By considering the pairwise coverage of diagrams, this number is already reduced to around 30 just by discarding test strategies that are duplicates in the sense that they guarantee the same coverage of pairs. By further applying the test suite minimization technique, the final numbers of tests are between 15 and 18.

We first evaluate the effectiveness of the proposed coverage criterion. Since the system is in its design stage and no implementation is currently available, we consider a synthesized correct implementation of the specification as a representative of the "correct" implemented system.

Inspired by mutation testing [14], we generate faulty implementations (i.e. mutants) by applying a combination of two categories of mutation operators on the synthesized implementation. The first type of faults simulates an incorrect comprehension of the specification and consists of randomly changing the events associated to transitions. The second type of faults is obtained by randomly removing transitions from the correct implementation, which simulates the possibility of forgetting potentially critical hidden obligations in the specification. The procedure of generating a mutant takes as input an *injected faults* rate, that is, the percentage of transitions, over the number of transitions in the correct implementation, that are modified by applying the mutation operators. This rate determines how "faulty" the mutant is and simulates the number of bugs that can be found in an implementation. A mutant is then generated by randomly choosing a transition and applying alternatingly one of the two mutation operators until reaching the given injected faults rate. The random choice of a transition follows a uniform distribution and does not allow multiple mutations to be applied on the same transition. Finally, we verify that the generated mutant is indeed an incorrect implementation, that is, we check if it violates the specification.

A naïve approach for detecting all possible faults of a mutant would be to execute all possible test strategies extracted from the specification. However this is often impractical especially when the specification is large and when the execution of each test is expensive since it requires the manual intervention of an operator. This is the case of the server maintenance system where the cost of executing a test is high and the number of all generated test strategies is around 1200.

To evaluate the effectiveness of the approach against a reduced number of tests we compare the tests generated by it to a set of test strategies chosen randomly out of the Global Test Strategy Graph. With this approach, the size of the set of generated random tests is the same as the one provided by our approach. Random test strategies are extracted by uniformly choosing one transition for each system state that is visited during a top-down traversal of the GTSG. Note that the randomly chosen tests are already specification-based, rather than being "purely random" ones. Nevertheless, we still observe significant improvement in bug detection ability when the tests are chosen based on the interaction coverage criterion rather than randomly.

Table I shows the results of the experiments for the three features of the system with three different values of parameter t of the t-wise active diagram coverage criterion: pairwise, three-wise, and four-wise. Each experiment has been executed 100 times, with different injected fault rates, and compares the average percentage of failed tests obtained by our approach with that obtained by the randomly chosen test-suite. A high percentage means a better capability of discovering faults. For a fair evaluation, the randomly chosen tests have the same number of states as the tests generated by the different coverage criteria.

As the table shows, our criterion generally outperforms a

TABLE I
TEST SUITE EFFECTIVENESS ON MUTATED IMPLEMENTATIONS, COMPARED AGAINST THAT OF RANDOMLY SELECTED TESTS

		Size of pair-wise		Size of three-wise		Size of four-wise	
operation	Injected Faults	Random Test Suite	PairWise	Random Test Suite	Three-Wise	Random Test Suite	Four-Wise
nodeAdd	1%	0%	76%	0%	88%	0%	94%
	5%	29%	88%	35%	88%	49%	94%
	10%	41%	94%	49%	94%	61%	98%
nodeRepair	1%	17%	56%	26%	89%	31%	89%
	5%	100%	100%	100%	100%	100%	100%
	10%	100%	100%	100%	100%	100%	100%
nodeUpgrade	1%	0%	11%	14%	44%	19%	65%
	5%	50%	94%	53%	94%	57%	94%
	10%	100%	100%	100%	100%	100%	100%

TABLE II
RUN-TIME EVALUATION OF THE APPROACH

operation	pairwise	three-wise	four-wise
nodeAdd	6339 ms	9503 ms	9267 ms
nodeRepair	6922 ms	9231 ms	10360 ms
nodeUpgrade	7048 ms	9232 ms	9184 ms

criterion in which a test is randomly generated. For example, for the case of 1% injected faults on the nodeUpgrade specification, the pairwise, three-wise and four-wise based test suites achieve 11%, 44% and 65% effectiveness, respectively, while the randomly chosen test suites of the same respective sizes achieve only 0%, 14% and 19%. This demonstrates the added value obtained by the wise interaction-based choice of tests. Note the increase in the effectiveness as the level of interaction grows, and the increase in the difference between our approach and randomly chosen tests.

Another observation to be made from this table is the growth in the added value as the percentage of faults drops. The cases of low fault percentage, where the faults are more rare and thus harder to detect are the ones in which the difference between the effectiveness of our interaction-based approach and that of the random choice is the most significant, and also the cases where the impact of the level of interaction, t, is the highest.

We also conducted experiments to evaluate the execution time for synthesizing test suites. We run the experiments on a MacBook with a 2,26 Ghz Intel Core 2 Duo CPU, and with 8 GB of RAM. Table II shows the run time of the experiments for the different specifications. We measure the total execution time for three different t-wise coverage criteria. The measured execution time for the pairwise coverage is around 7s and between 9s and 10s for the three-wise and four-wise coverage. The execution time of random tests is also around 7s. The results shows that our approach is also efficient and scalable in terms of performance, especially if compared to the time required to manually derive the tests.

While these results are highly encouraging and promising, a more thorough evaluation is still required—this is left as future work at this point.

VI. RELATED WORK

Deriving test cases from specifications is not a new idea. Several approaches exist that consider scenario-based specifications as the basis for deriving tests [21], [16], [3], and many approaches use scenarios as a means of specifying tests themselves [9], [8], [19], [22], [20], [17].

However, our work is novel in three aspects. First, being based on MSDs, our approach is based on a scenario specification language that allows engineers to formally define safety and liveness requirements and which precisely defines the concurrent activation and progress of scenarios. Other scenario specification languages often only support the specification of possible scenarios [3], [16] or the interaction of scenarios has to be specified explicitly [21]. Second, we define a new coverage criterion for tests for the combinatorial coverage of concurrent scenario activations, inspired by CTD. To the best of our knowledge, this idea has not been investigated before. Third, we propose an automated synthesis method for generating test strategies, which can also deal with possible non-determinism in early scenario-based specifications.

In the following we report on selected related approaches.

Cartaxo et al. [3] automatically generate tests from UML sequence diagram specification. The specification is translated into a labeled transition system from which tests are then extracted by traversing all its paths. Their usage of UML sequence diagrams, as opposed to MSDs, limits the specications' expressiveness and they present no coverage criterion.

Lee et al. [16] also consider generating tests from sequence diagrams. Sequence diagrams are translated into a state machine representing changes in the system's state variables. Tests are then derived by requiring node coverage or edge coverage (state tour and transition tour, resp.) of the global state machine. This will create many tests, limiting the scalability, as opposed to our approach, which tries to cover only t-wise scenario coverage.

Similar to the previous approach, Briand et al. [2] generate test cases from control flow graphs created from sequence diagrams and state machines. However, when generating tests, only one sequence diagram is considered at a time, thus neglecting the concurrent occurrence of scenarios.

Nebut et al. [18] propose an approach where specifications are given as use case descriptions. A transition system can be derived from the use cases when steps are enriched by pre- and post-conditions. From the resulting model, test cases can be derived. Use case interactions are, however, not regarded in the test coverage.

Many works consider different forms of scenarios as a means to specify the tests themselves, rather than to specify the behavior of the system [9], [8], [17]. For example, TESTOR [19] is a system for generating test sequences from behavioral models. A similar approach is taken by Pickin et al. [20], where a system is modeled by UML class, state, and object diagrams, and sequence diagrams are used as test objectives, which guide the test generation algorithm.

VII. Conclusion

We propose a novel approach for the automatic generation of test strategies based on an MSD specification with a new test coverage criterion for measuring the amount of concurrent scenario activations tested. The key idea of the criterion is inspired by Combinatorial Test Design. Given an MSD specification, our synthesis technique can automatically synthesize a minimal set of test strategies that exhibits a maximal t-wise scenario interaction coverage.

The technique could be transferred to other scenario-based specification approaches if they provide a formal semantics for concurrent scenario activations, or also to LTL specifications if they follow a scenario-based specification style.

We evaluated our approach with an example based on an industrial specification. Our evaluation shows that our synthesized test suites are more effective than randomly chosen tests. The preliminary positive feedback received by our industrial partner also suggest the effectiveness of our technique and its applicability in a real context.

As current and future work we plan to evaluate the quality of the produced test suite on the basis of real, human-engineered system implementations. We plan also to extend our technique by investigating and evaluating the effectiveness of other coverage criteria based on MSDs. Moreover, we will investigate how the t-wise coverage criterion can be adopted for measuring the quality of existing test suites.

As future work, we also plan to target the scalability of the approach. The size of the game graph is exponential in the number of scenarios. Moreover, in some applications we may have to consider message parameters with large value ranges, where enumerating all values will be impractical. Here, a symbolic interpretation of these parameter values in the game graph could be a solution.

References

[1] C. Brenner, J. Greenyer, and V. Panzica La Manna. The ScenarioTools play-out of modal sequence diagram specifications with environment assumptions. In *Proc. 12th Intl. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2013)*. EASST, 2013.

[2] L. Briand, Y. Labiche, and Y. Liu. Combining UML sequence and state machine diagrams for data-flow based integration testing. In *Proc. 8th European Conference on Modelling Foundations and Applications*, ECMFA'12, pages 74–89, Berlin, Heidelberg, 2012. Springer-Verlag.

[3] E. Cartaxo, F. Neto, and P. Machado. Test case generation by means of UML sequence diagrams and labeled transition systems. In *Proc. Intl. Conference on Systems, Man and Cybernetics*, pages 1292–1297, 2007.

[4] F. Cassez, A. David, E. Fleury, K. G. Larsen, and D. Lime. Efficient on-the-fly algorithms for the analysis of timed games. In M. Abadi and L. de Alfaro, editors, *CONCUR 2005 Concurrency Theory*, volume 3653 of *LNCS*, pages 66–80. Springer Berlin Heidelberg, 2005.

[5] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-Based Testing in Practice. In *Proc. 21st International Conference on Software Engineering (ICSE'99)*, pages 285–294. ACM, 1999.

[6] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. In *Formal Methods in System Design*, volume 19, pages 45–80. Kluwer Academic, 2001.

[7] A. David, G. Behrmann, P. Bulychev, J. Byg, T. Chatain, K. G. Larsen, P. Pettersson, J. I. Rasmussen, J. Srba, W. Yi, K. Y. Joergensen, D. Lime, M. Magnin, O. H. Roux, and L.-M. Traonouez. Tools for model-checking timed systems. In O. H. Roux and C. Jard, editors, *Communicating Embedded Systems – Software and Design*, pages 165–225. ISTE Publishing / John Wiley, Oct. 2009.

[8] M. Ebner. Ttcn-3 test case generation from message sequence charts. In *Workshop on Integrated-reliability with Telecommunications and UML Languages (ISSRE04:WITUL*, 2004.

[9] F. Fraikin and T. Leonhardt. SeDiTeC – testing based on sequence diagrams. In *Proc. 17th IEEE Intl. conference on Automated software engineering*, ASE '02, pages 261–266, Washington, DC, USA, 2002. IEEE Computer Society.

[10] J. Greenyer. *Scenario-based Design of Mechatronic Systems*. PhD thesis, University of Paderborn, 2011.

[11] J. Greenyer, C. Brenner, M. Cordy, P. Heymans, and E. Gressi. Incrementally synthesizing controllers from scenario-based product line specifications. In *Proc. 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 433–443, New York, NY, USA, 2013. ACM.

[12] D. Harel and S. Maoz. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software and Systems Modeling (SoSyM)*, 7(2):237–252, May 2008.

[13] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, August 2003.

[14] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678, Sept 2011.

[15] D. R. Kuhn, D. R. Wallace, and A. M. Gallo. Software Fault Interactions and Implications for Software Testing. *IEEE Transactions on Software Engineering*, 30:418–421, 2004.

[16] N. H. Lee and S. D. Cha. Generating test sequences from a set of MSCs. *Comput. Netw.*, 42(3):405–417, June 2003.

[17] S. Maoz, J. Metsä, and M. Katara. Model-based testing using LSCs and S2A. In A. Schürr and B. Selic, editors, *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 301–306. Springer Berlin Heidelberg, 2009.

[18] C. Nebut, F. Fleurey, Y. Le-Traon, and J.-M. Jezequel. Automatic test generation: a use case driven approach. *Software Engineering, IEEE Transactions on*, 32(3):140–155, 2006.

[19] P. Pelliccione, H. Muccini, A. Bucchiarone, and F. Facchini. TESTOR: deriving test sequences from model-based specifications. In *Proc. 8th Intl. Conference on Component-Based Software Engineering*, CBSE'05, pages 267–282, Berlin, Heidelberg, 2005. Springer-Verlag.

[20] S. Pickin, C. Jard, T. Jeron, J.-M. Jezequel, and Y. Le Traon. Test synthesis from UML models of distributed software. *IEEE Trans. Softw. Eng.*, 33(4):252–269, Apr. 2007.

[21] J. Ryser and M. Glinz. Using dependency charts to improve scenario-based testing – management of inter-scenario relationships: Depicting and managing dependencies between scenarios. In *Proc. 17th Intl. Conference on Testing Computer Software, TCS 2000*, 2000.

[22] M. Satpathy, Q. Malik, and J. Lilius. Synthesis of scenario based test cases from B models. In K. Havelund, M. Núñez, G. Rosu, and B. Wolff, editors, *Formal Approaches to Software Testing and Runtime Verification*, volume 4262 of *Lecture Notes in Computer Science*, pages 133–147. Springer Berlin Heidelberg, 2006.

[23] K. Tai and Y. Lie. A Test Generation Strategy for Pairwise Testing. *IEEE Transactions on Software Engineering*, 28:109–111, 2002.

[24] K. Tatsumi. Test-Case Design Support System. In *Proc. Intl. Conference on Quality Control (ICQC)*, pages 615–620, 1987.

[25] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.