Framework for Flexible, Adaptive Support of Test Management by Means of Software Agents

Claudius V. Jordan[1], Franziska Mäurer[1], Sven Löwenberg[2], and Julien Provost[1]

***Abstract*—In the context of system testing on Hardware-in-the-Loop (HiL) test benches, performing functional tests on electrical control units (ECUs) comes with various challenges. Especially planning the execution of test cases for a test cycle and nightly batch runs is cumbersome. In order to support the test managers, this work proposes an adaptable and flexible test management assistance system. Therefore, a set of requirements for such an assistance system have been identified and proposed solutions from the literature have been recalled. Out of those, software agents appear to be a suitable method for building up the framework of the assistance system. The framework proposed in this paper is divided into two phases: static prioritization of test scripts and dynamic execution ordering. The agents of the former phase aggregate relevant available data from various sources to calculate a test script priority. In the latter the agents negotiate to solve the resource allocation problem for test scripts on test benches. In addition, test logs are taken into account at run-time for reprioritization. Finally, an industrial use case from the automotive industry is taken as an example to discuss the applicability and current obstacles on the way to application and qualitative evaluation of the expected benefits are discussed.**

***Index Terms*—Agent-Based Systems; Planning, Scheduling and Coordination; Task Planning**

I. Introduction

THE organization of testing activities and in particular the execution planning for nightly execution are based on experienced test managers. They combine a multitude of aspects and information from various sources (including informal ones) in order to come up with a somewhat *optimal* test execution plan. One conversant example is system testing in the automotive domain, where multiple ECUs (electronic control unit) are combined and the aim of testing is the assessment of the fulfillment of required functionality, it turns out to be a challenging endeavor. The increase of software within automotive environments and the subsequent increase in complexity puts further strain on operators. The multitude of functionalities within a system requires many test cases, while the available testing resources, i.e. Hardware-in-the-loop test benches (HiL), are limited. This is particularly troublesome when the same testing resources are shared for various tasks, i.e. commissioning of automated test scripts during the day and their execution during the night. Note that we call *test scripts* executable programs that represent requirement-based test cases, that consist of a sequence of several action steps. Hence, using the available testing time as effectively as possible increases the amount of executed test scripts and, consequently, the chance of finding more faults in the system under test (SUT).

Manuscript received: February, 14, 2019; Revised April, 15, 2019; Accepted May, 7, 2019.

This paper was recommended for publication by Editor Jingshan Li upon evaluation of the Associate Editor and Reviewers' comments.

[1]Assistant Professorship for Safe Embedded Systems, Technical University of Munich, Germany claudius.jordan@tum.de

[2]Trace Tronic GmbH, Dresden, Germany

Digital Object Identifier (DOI): see top of this page.

A. Challenges

The main challenges test managers face is that manually deciding what an *optimal* test execution plan would be is almost impossible. As the scheduling and prioritization of test scripts is done manually, the test manager is supposed to use the testing time effectively, and at the same time consider multiple aspects e.g. the required precondition system states of test scripts successively executed, test bench configurations, dependencies between test scripts and test actions potentially leading to dependencies in the failure behavior.

Further challenges include gathering all necessary information like changes in subsystems, requirement specifications or bug fixes. The information is distributed among multiple resources and might not be available in a structured document or database but rather as tacit knowledge of specific experts which might or might not be involved in the testing process, e.g. developers. That already makes acquiring relevant information as such a difficult venture considering that it is distributed among multiple resources including informal discussions. Hence, it is an additional challenge for defining an *optimal* prioritization order given the fact that not all necessary information is available.

Also the test environment itself is under constant development or sometimes certain resources are unavailable, which leads to the additional difficulty of reacting to the ongoing changes. This results in additional constraints that need to be considered in the execution order, in particular when there is no test engineer available during the automatic execution.

During the system testing phase, HiLs offer a flexible and easily adaptable test environment [1]. For example in the context of vehicle on-board supply systems, such HiLs are composed of various batteries and physical ECUs under test embedded into mock-ups and simulation of unavailable system parts. Test scripts are composed of three blocks: pre-processing (setup), action and post-processing (teardown). Whereas during pre-processing a HiL adopts a certain system state, from which the actual requirement-based test is performed, post-processing is necessary for the HiL to adopt a well-defined system state (e.g. reset to idle) such that a subsequent test script can be started. Note that due to their

structure, all test scripts are self-contained and designed to be independently executable. Thus, they can be executed in any order and,apparently, the order of test cases has a significant influence on the overall execution time if there are inert system state parameters (such as the state of charge of a battery in our use case).

To the best of our knowledge there exists no solution that permits to fully tackle the the aforementioned issues. More details on related work that partially address these challenges are given in Sec. II.

B. Requirements

In order to develop a flexible, adaptive test management assistance system, and to address the identified challenges, the following set of requirements originate from discussions with several test engineers from the industrial partner.

R1 - Prioritization of Test Scripts: With the increasing amount of test scripts and limited testing time, the execution of all available test scripts in one development cycle is a challenging endeavor, which is still done mostly manually. Test scripts that are more valuable should be prioritized. To support this process, the assistance system should be able to determine a value for each test script based on several sources of information, whose availability might vary over time. As the framework should be applicable to a variety of test projects, where the available information varies in quality and importance, the framework should provide a suitable means for customization to the test managers' preferences.

R2 - Scheduling of Test Scripts: In the context under consideration, various variants of system configurations are considered and there exists a link between test cases and those configurations. Also, test cases require different system states for their execution. Given that there are multiple configurable test benches for which changing the configuration and system state is time-consuming (if it is possible at all to automatically change the configuration without the need for human intervention), the challenge is to assign test scripts to suitable test benches considering that, depending on the test bench's current system state, the preprocessing for test scripts varies. Therefore, the assistance system should consider this adaption and reconfiguration time.

R3 - Dynamic Adaption to Test Logs: It is possible during testing that test system misbehavior results in failures other than discovering actual functional defects. Since the test execution generally is unsupervised, no operator is available to handle such a case. Several cases should be considered. Based on the hypothesis that a failure in one test case can lead to failing similar ones too, a waste of time could be prevented when dependent test cases on the failing one are not executed. However, if a failure is encountered, it might be useful to provoke similar failures with the help of similar test cases in order to provide as much information as possible to the test experts. Either way, based on the specific project at hand, the assistance system should incorporate a dynamic adaption strategy that considers new test logs acquired during run-time for the scheduling.

R4 - Interaction with the Test Manager: Although the framework should be able to act as independently as possible, the knowledge and experience of the test manager might be essential. This could be the case in particular when information is unavailable or decisions have to be taken. Therefore, the assistance system should allow interaction with the test manager. Interaction comprises inquiring for data or decisions, and displaying information on demand.

In the following, frameworks addressing similar challenges found in the literature are discussed in Sec. II. There, the concept of software agents is introduced and related works regarding test case prioritization and scheduling are recalled. In Sec. III, the proposed concept of the flexible, adaptive test management system is presented in detail. Afterwards, the applicability of the proposed concept to a specific industrial use case is evaluated in Sec. IV and discuss further application and development directions in Sec. V.

II. Background and Related Works

Based on the research questions (Sec. I-B), this section evaluates related works. First, software agents and their use in the context of testing are discussed (Sec. II-A), then other approaches are presented addressing test case prioritization (Sec. II-B) and scheduling (Sec. II-C). Finally, we introduce a simplified test model used for data aggregation (Sec. II-D).

A. Software Agents in Testing

Software agents, hereafter referred to as agents, are suitable for decentralized, structurally variable and cooperative systems, which have already found application in testing [2], [3]. Various definitions for agents exist but in our work we adhere to the view of Wooldridge and Jennings [4], where an agent is seen as a goal-oriented, autonomous system that is able to interact (sense, manipulate and communicate) with the environment in which it is situated. There are two ways to decompose a problem into agents: functional or physical decomposition [5]. Whereas functional decomposition encapsulates certain functionalities in an agent, physical decomposition distributes agents among physical components resulting in a direct relationship of an agent and its associated physical component. Approaches distributing different tasks, e.g. the execution or the analysis of test cases, among agents are proposed in [6]. The drawbacks of functional decomposition are the high amount of shared state variables leading to unintended interactions and inconsistencies [5]. Exemplary approaches using physical decomposition are [2], [3]. Both consider test case prioritization; however, test case scheduling among test resources and test case dependencies are not considered. In [7], the authors suggest an additional negotiation-based test case scheduling. Test case agents allocate themselves by negotiating with the test resource agents. Even though physical decomposition defines distinct state variable sets and thus enables only limited interactions between agents, the distribution among physical components can quickly lead to a complex management of agents and a communication overhead [5].

B. Test Case Prioritization

Test case prioritization is a broad field of research. The authors in [8] performed a literature survey to identify the main utilized influencing factors of prioritization approaches out of which code and requirement coverage are the most commonly chosen. Further frequently used approaches are based on the execution time, the fault coverage and historical data. They also identified that the combination of multiple prioritization factors is a vital aspect for an efficient and effective test case prioritization strategy.

Apart from coverage, several other requirements-based approaches have been proposed out of which a few are cited here. For example, in [9], requirements are weighted based on the amount of changes, the customer importance and the complexity of requirements, which involves considerable manual effort. In [10], dependencies between requirements are used, allowing change propagation considerations.

Besides dependencies between requirements, other relations between artifacts have also been used. In [11], for example, dependencies between test cases, targeted software modules and their underlying requirements are constructed. Similarly, a system model and a test case model are related in [12]. More elaborate ontologies have been proposed recently e.g. in [13], which provide dependencies between various sorts of information present in test processes.

The system model contains the dependencies between test benches, test cases, components and functionalities. The test case model contains dependencies of test cases and their respective test actions. Based on the test importance of test components and the coverage of test actions, the prioritization is calculated. Prerequisites of the approach are historical test data, thorough knowledge of the test benches and their respective components and functionalities and circumstantial version management to know the changes of each release. The authors in [14] prioritize test cases based on their dependencies degree, which was manually elaborated. Another idea is to apply data mining approaches for similarity measurements of test cases. That way redundant and dependent test cases can be identified. Several methods have been proposed relying on similarity measurements [15], [16], [17]. There, test cases are treated as continuous strings and different distance metric methods are applied to measure the similarity between test cases.

A common approach derives the prioritization order by weighting the prioritization factors to obtain a priority [18]. A more complex approach employs fuzzy logic to represent the expertise of the test manager and derive priorities [3], [19]. The approach proposed by [20] considers the interference of the prioritization factors. Subjective weighting is utilized to calculate weights based on neighboring importance of the objectives. Each objective is weighted based on the importance compared to the other objectives. Even though the evaluation shows the effectiveness of the proposed approach, a time-consuming evaluation of the factor relations would be needed in advance.

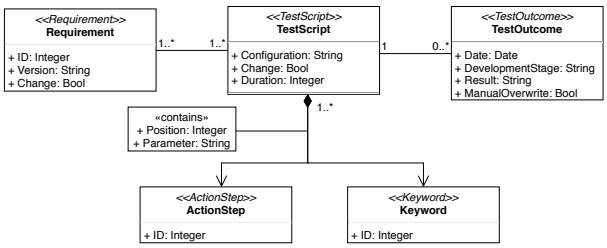

Fig. 1. Test model for information aggregation

C. Test Case Scheduling

Test case scheduling addresses the challenge to find a suitable test case for each test resource or vice versa. In [21], this problem is tackled employing constrained programming. Even though this approach provides promising results for static distribution, dynamic distribution, i.e. new distribution based on test results during the execution, is not considered. In the field of shop floor production processes, agent-based scheduling approaches are proposed. Instead of distributing test cases on several test resources, different product tasks are distributed among multiple production systems. Agents can reduce product lateness by providing a dynamic negotiation scheme. The authors in [22] present a dynamic scheduling approach for Cyber-Physical Production Systems using the Contract Net Protocol. They chose this protocol as it is a mature mechanism due to its wide use and continuous development. In [23], the authors propose a semi-centralized approach extending the Contract Net Protocol. Compared to a negotiation between an initiator and participants, the extension introduces an additional manager. This manager limits negotiation by proposing suitable participants to the initiator with whom he can communicate.

D. Test Model

To aggregate the needed information and to tackle the problem of data preparation, we establish a test model with the modeling language UML. Our approach is inspired by the definition of system and test models as proposed in [11], [12]. Therein, the authors modeled test scripts and their respective test actions to aggregate information about test script dependencies. A simplified version of the test model is displayed in Fig. 1, which only shows the dependencies between requirements, test scripts, their containing test steps and test results. A detailed test model is out of the scope of this article introducing the framework, however a more elaborate version geared to the ontology presented in [13] will be considered during quantitative evaluation of the actual implementation.

III. The Flexible and Adaptive Test Management Assistance System

The overall goal of the assistance system is an effective test management. The system is a multi-agent system, which can be divided into two phases (Fig. 2). The first phase deals with test data from previous test cycles (Sec. III-A). An interface for each resource extracts the relevant information. From this,

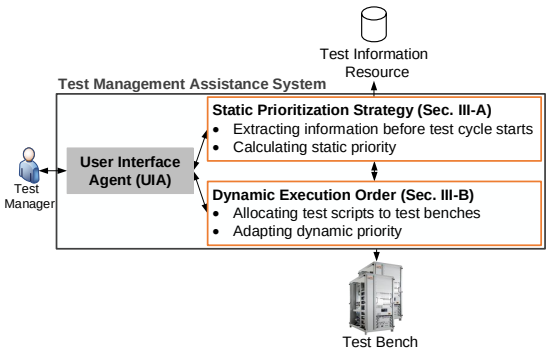

Fig. 2. Overview of the assistance system

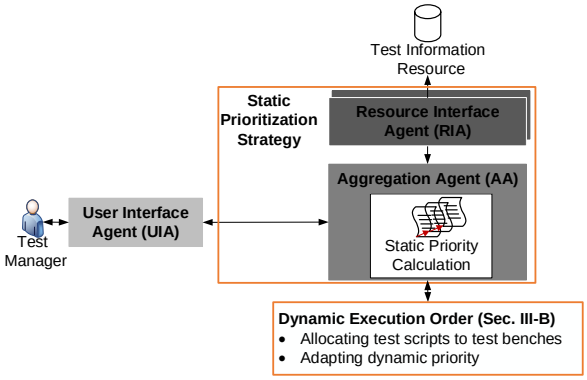

Fig. 3. Static prioritization strategy of the assistance system

a static priority is calculated to assess the value of a test script. Based on the static priority, the second phase determines an execution order for each test bench (Sec. III-B). Agents allocate test scripts to test benches for execution and based on the test logs acquired at run-time they dynamically adapt the test script priority.

Additionally, the assistance system requires an interface with the test manager (cf. R4). The test manager can intervene in the execution process and provide his expertise, e.g. information about how to proceed with failed results. We realize this through a *User Interface Agent* (UIA), which reports the results to the test manager (cf. Sec. III-C).

A. Static Prioritization

To determine a priority value for each test script, a set of *offline* agents extract available information from adjacent data stores. In the following, we describe briefly the different agents and their tasks, which are displayed in Fig. 3. The *Resource Interface Agents* (RIAs) establish the communication to and extraction of information from adjacent systems, i.e. data stores. They also monitor their respective target for changes. We adopt the idea from the concept of interface agents in [2] as this allows fast adaptability to resource changes. For informal and undocumented information, the UIA takes over for the communication to and information gathering from the test manager. The *Aggregation Agent* (AA) then aggregates the information collected by the RIAs and UIA and calculates the static priority value.

In accordance with the findings in the literature, we identified four essential factors that play a role in prioritizing test cases in functional testing that are considered for static priority: namely, manual preference from the experienced test manager, requirement changes and subsequent updates of test scripts, historical data, and component changes. Fig. 4 shows the factors and how they relate to each other. The Manual Priority (MP) allows consideration of the valuable expertise of the test manager (cf. R4). The Requirement Changes Priority (RCP) comprises the importance of test scripts induced by requirement changes. It also considers dependent requirements and potentially necessary subsequent test script adaptions. Consequently, those updated test scripts should be somewhat more important than others. This might also vary depending on

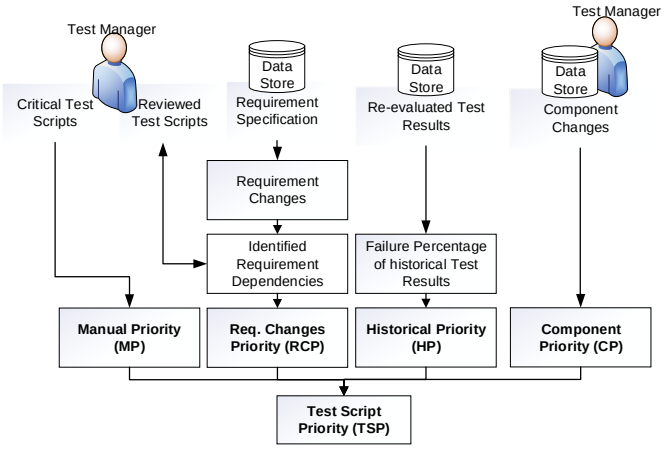

Fig. 4. Prioritization strategy

the severity of the requirement change, which again only the test manager can provide as an input. The Historical Priority (HP) integrates information about failure detection abilities of test scripts. Therefore, a well-tended database of past test executions and their outcomes and special occurrences is indispensable. Error origins of failed tests must be identified and registered. This allows excluding test system defects from the priority calculation. The Component Priority (CP) examines changes of test bench components. If a component underwent some bug fixes for example, test scripts that are related to this component are considered more important than others. Depending on the availability of the change information, the information could be extracted from a data store (e.g. version management system) or is again requested from the test manager. A prerequisite for the CP is a component model and a mapping between components and test scripts and also the severity of changes might influence the CP value.

Information Aggregation: Based on the acquired information, the Test Script Priority (TSP) can then be calculated by the AA. An intuitive approach is to calculate the weighted sum of the aforementioned factors but other approaches applying fuzzy logic are also imaginable (cf. [7]). The presented approach allows including the test manager's knowledge about the reliability and importance of each factor (cf. R4). An advantage of the flexible framework we propose is that it is

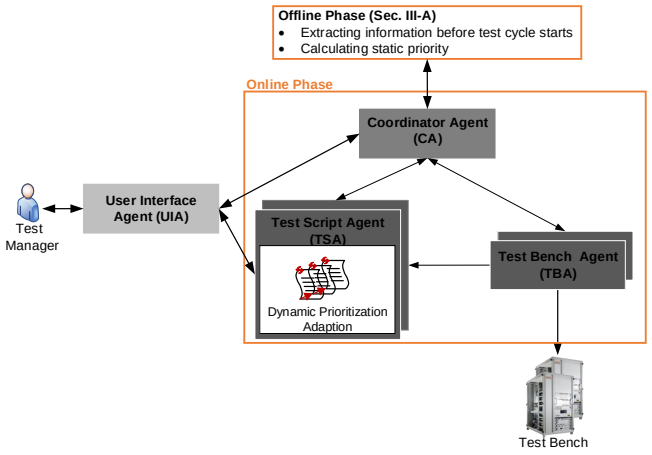

Fig. 5. Dynamic execution ordering of the assistance system

easy to consider additional influences on the given factors, replace test manager inquiries by rule-based automatic procedures, or even add new factors. One thinkable extension is to include a separated priority factor for system states or test script attributes.

B. Dynamic Execution Ordering

To determine a dynamic execution order, a set of *online* agents allocate test scripts to test benches and dynamically adapt their priority value based on test results. We describe briefly the different agents and their tasks, which are displayed in Fig. 5. Here, the physical decomposition allows an agent to encapsulate specific goals and knowledge, and enables unambiguous responsibility for a limited amount of variables [7], which will prove to be beneficial.

One *Test Script Agent* (TSA) is assigned to each test script, which store relevant metadata (cf. test model Sec. II-D), i.e. the static priority calculated beforehand. Furthermore, they are responsible for the communication and negotiation with other agents. Particularly, this includes dynamically adapting the test script priority at run-time as discussed in Sec. III-B2. Note that it is also up to the TSA to reduce the priority to 0 once the test script has been executed successfully if it is not required to be executed again.

Test Bench Agents (TBAs) represent the interface to the test benches similar as the RIAs do for data resources. This is beneficial to ensure the flexibility especially for evolving test environments, when the availability of resources changes frequently. The TBAs monitor system state and status of their respective test bench and are responsible for executing test scripts. The overall goal of TBAs is maximizing the time executing valuable test scripts. This means finding a trade-off between adapting the system state required by test scripts in order to execute them and the preparation time it takes to do so. This effect is discussed in Sec. III-B1.

An additional *Coordination Agent* (CA) organizes the test script allocation process. The CA bounds the communication between the TBAs and TSAs and, thus, is expected to optimize the time performance as shown in [23]. Setting a limit to the maximum number of communicating agents mitigates the risk of a communication overhead [5] due to a project-specific and possible high amount of test scripts and test benches. The UIA again takes over the communication with the test manager.

1) Negotiation-based Test Allocation: This section addresses the introduced scheduling problem motivated in R2. The static and dynamic priority as well as the system state and available execution time are considered for an effective execution ordering. The proposed approach is inspired by the proposal to apply negotiation mechanisms from [7].

A test cycle begins with a new development stage detected on a test bench by the TBA. To assign the static priorities, the CA communicates with the AA and instantiates the TSAs. The proposed negotiation protocol (Fig. 6) is a modification of the Contract Net Protocol extended with a coordinator from [23]. For simplicity, the communication is limited to TSAs and TBAs with a matching configuration.

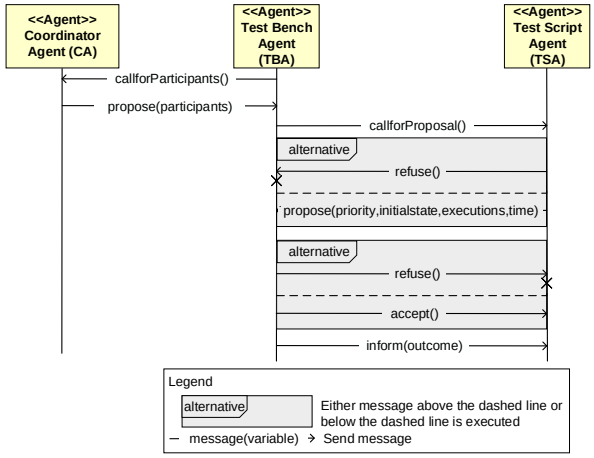

Fig. 6. Negotiation protocol modeled in agentTool

The negotiation process starts when a new test session is started. Initially the TBAs request the CA for participants to find a set of suitable test scripts. The CA proposes a list with TSA containing a matching configuration, based on which TBAs call for proposals of the respective TSAs. A TSA refuses a proposal from the TBA for example in case it is currently executed on another test bench. Otherwise, if a TSA accepts, it sends an offer including its priority value and required system state. From all received offers, the TBA determines the test script to be executed, finding the best trade-off between the highest priority and the lowest required system state adaption time. Thereafter, the TBA executes the test script and sends the outcome back to the TSA.

2) Dynamic Prioritization Adaption: Based on the test logs that a TSA receives from a TBA, it calculates a new dynamic priority and informs other TSAs about special occurrences, i.e. failed test actions. Hence, all dependent TSAs can adapt their dynamic priority.

Consequently, the dynamic priority allows reacting to test logs at run-time, which is motivated in R3. This is particularly beneficial in the case that a certain test action fails. A test action might represent an interaction with the SUT, e.g. a

message is sent to a certain ECU. Now, it is possible that the test action itself fails, e.g. the communication channel is unavailable, leading to inconclusive test results. As a result, other test scripts that contain the same test action could be affected, which we account for by adjusting the prioritization. This is beneficial to prevent unnecessary test executions (with potentially inconclusive results) and can improve the test execution efficiency [14]. Especially in the exemplary case described here, a high probability of failure can be predicted for other test scripts containing the same test action. Thus, time is saved by not executing those dependent test scripts. On the other hand, a dynamic prioritization can help to localize faults. Executing test scripts with the same test action or executing the respective test script on other test benches can help localize the failure origin by providing more (potentially) relevant information to the test engineers. As a consequence, an appropriate dynamic prioritization strategy is highly dependent on the project and the current status in a test cycle.

For an error-prone test bench, the execution of a test script dependent on failed ones may not be beneficial. As it is likely that the test bench is the error origin, executing test scripts with a high likelihood to fail would most probably result in wasted execution time. However, a focus on localizing faults might help to identify and fix those faster. Hence, it could be more appropriate to execute the test script and depending test scripts again on test benches with different configurations to narrow down the error origin. Since up until now only the test manager knows about existing defects and has the experience to assess the current status, we introduce reprioritization modes for them to chose from, which represent three different point of views.

(1) A failed test script should be executed on other test benches to support finding the error origin. The failed one should not be executed on the same test bench until the fault is identified. Hence, the test manager can reason about an error origin based on test logs of various executions on different test benches.
(2) A failed test script should be executed on other test benches. Meanwhile, dependent test scripts should be executed on the failed test bench[1] to support finding the error origin.
(3) The failed test script as well as all dependent test scripts should not be executed until the fault is identified. Thus, no execution time is wasted for test scripts with a high failure likelihood. However, the testers perform their investigation based on less information.

Depending on the chosen reprioritization mode, TSAs adapt the priorities accordingly. Note that it is possible to assign priorities depending on the test benches.

C. Reporting

At any time, a test manager can query the UIA in order to receive information about the current status. Particularly after the static prioritization (cf. Sec. III-A) and finally after the test execution run, the UIA provides a multitude of information. As the UIA does currently not provide a self-containing GUI, the information of interest can be accessed by querying the UIA. In the use case, existing tools are used for processing data or displaying relevant information to the test manager.

[1]The test bench on which the executed test script of interest has led to a fail verdict.

IV. Discussion

The presented approach of an agent-based framework for test management is discussed for an industrial use case. In this section, we describe the applicability and further investigation for an implementation in current test execution planning. The availability of all required data is checked and the required additional manual effort is assessed.

As described in Sec. III-A several data stores need to be available in order to allow effectively prioritizing test cases automatically. Particularly a test script design tool and a test result management tool are indispensable. In addition, communication with test benches and the databases are required for the dynamic execution presented in Sec. III-B. Note that the proposed framework is not dependent on specific tools but on the functionality already provided by test automation tools usually used in testing projects. In our use case, the software tools ECU-TEST[2], and the test result management tool TEST-GUIDE[3] are used. ECU-TEST contains the test script implementations and controls the test execution on test benches. The test logs are stored in a database that is accessed by TEST-GUIDE. Requirement specifications are available in a requirement management software. These specifications include all tested requirements describing the functionality of the vehicle on-board supply system. Consequently, the prerequisites for applying the framework to the use case are fulfilled. In the following the static prioritization is discussed first and then the dynamic execution order.

A. Static Prioritization

Test scripts follow the precondition (setup), action and postcondition (teardown) structure. Each of the blocks includes several sequential action steps. The pre- and postcondition blocks comprise several actions that lead the SUT (in fact the whole test bench system) into a desired state. The precondition block contains information about the test precondition state and how the SUT is brought to it. Similarly, the postcondition block resets the test bench to a state from which (any other) subsequent test script can be executed.

Each action step itself represents high-level interactions with the SUT, which can be divided into sub-procedures. A common term for such a high-level interaction used in the context of testing is *keyword* (or *block*), which contains the implemented action step procedure that in turn can contain keywords. Generally, keywords can be understood as function calls and as such they might depend on passed parameters. In addition, keywords are described with unambiguous comments at the top-level. The keywords are stored in the test model (Sec. II-D). From this, dependencies between test scripts can

[2]https://www.tracetronic.com/products/ecu-test/
[3]https://www.tracetronic.com/products/test-guide/

be automatically constructed, which are useful for deriving the requirement changes priority (RCP) and later during dynamic re-prioritization.

Each test script contains predefined attributes, containing the respective meta-data. Maintained attributes include the requirement specification ID, the estimated total execution duration, and the required vehicle type and battery configuration. Particular to the automotive environment are so-called diagnostic trouble codes (DTC), which also form part of the meta-data set. The requirement IDs are used to derive dependencies between the requirements and test scripts, which are useful for deriving the RCP. We leverage the other attributes later.

As mentioned in Sec. III-A, a historic priority (HP) can only be calculated if a well-tended database of past test executions is available. Even though a test result management tool is used, the data lacks the evaluation of failure causes. The prerequisite for calculating a meaningful HP is a re-evaluation and clearance of the historical data in order to eliminate all failure causes other than actual defects in the SUT. Deriving the component priority (CP) appears to be a particular challenge as up to now no component model for the test benches exists or is not kept up to date. Instead of using components, in the project at hand they decided to form clusters of test scripts with respect to so-called test topics. Unfortunately, there is no automatic process in place nor proper documentation that would allow determining changes with respect to the test topics, which results in manual consultation of responsible experts, aggregating the information and assigning a priority value to each test topic manually. Fortunately, this is feasible in the proposed approach.

B. Dynamic Execution Order

Up until now, at the end of a work day, test experts collect test scrips in a list such that they are performed sequentially one test case after another. The test scripts are selected and ordered based on experience and maybe some rules of thumb. The test bench agent (TBA) makes use of the precondition information when it determines the required time for a system state change. A first guess for the expected change duration is the sum of the duration of each test action in the precondition, which apparently depends on the current system state in particular for inert system state parameters like the state of charge of a battery. Furthermore, the TBA could forecast the postcondition state and initiate negotiations with TSAs for subsequent executions in advance and parallel to the current execution, which is currently not considered in the framework.

In the project at hand, it is the case that sometimes a component of the test environment fails, which leads to many unsuccessful test executions. In other words, once the component fails, all subsequent test scripts in the list fail, resulting in wasted execution time and diagnostic effort for test engineers. Thus, it is considered beneficial to apply the negotiation-based, dynamic allocation procedure proposed in this paper in order to react to such component failures adequately. But given the fact that up until now failures are not thoroughly evaluated and no information about possible failure causes are kept in the data base, the dynamic prioritization cannot be applied yet. On the other hand, by experience the DTC is helpful to localize faults, e.g. in case a test script execution fails, it is helpful to execute other test scripts with the same DTC. Consequently, an additional data store with special information about DTCs could prove useful for the re-prioritization.

C. Reporting

According to R4 (cf. Sec. I-B, it is not only important that the test manager can provide important (and necessary) information for the static prioritization via the UIA but it is equally important that the UIA reports relevant information back to the test manager in a suitable way. As mentioned in Sec. III-C the UIA does not provide a GUI itself but allows to connect to existing tools, such as the test result management tool used here.

D. Qualitative Expected Benefits

The presented framework permits to reduce test runs with inconclusive results (cf. Sec. III-B2), which in turn leads to a reduced time of analysis and result interpretation after nightly batch runs. The test engineers working actively on the project the use case is based on, stated that they spend on average 2 hours per day reevaluating failed test runs and consequently on test execution planning preparing the next nightly batch run. One test manager responsible for one system aspect (out of various), for which around 200 test cases exist, stated that she spends between 2 and 3 days selecting the test cases which should be executed (sometimes multiple times) in an upcoming test cycle, which last for 4 to 6 weeks. On such lengthy test cycles, being able to react dynamically to inconclusive tests is of particular interest. In terms of potential hours saved, the expected benefits of our approach can be estimated as at least half of the total hours spent on test case selection and execution planning by the proposed framework, namely static prioritization and dynamic execution ordering. In addition, it is expected that by reprioritizing test cases as reaction to test outcomes within the nightly batch run, less inconclusive test runs will be observed leading to additional saving in terms of reevaluating work. Note that the expected benefits presented here are in fact general to all testing processes that are faced with manual test selection and execution planning.

Within this section it is shown that the framework is applicable to an industrial setting with certain boundary conditions due to availability of data. Despite the unavailability of for example a component model, the test script priority (TSP) can be calculated with the help of additional information provided by the test manager. On the other hand, additional information sources have been identified, e.g. DTCs, which can easily be integrated into the data aggregation and thus extend the proposed framework. It is crucial for the static prioritization to set up a well-tended database and re-evaluate the test results if this information is to be considered. Otherwise again a test engineer would be needed to manually provide the information. Overall, a substantial amount of manual work is still necessary in that use case in order to fully exploit the agent-based adaptive test management assistance system,

however experts in the automotive field state that a substantial amount of hours can be saved when having such an assistance system in place.

V. Conclusion

In this paper, we propose an agent-based test management assistance system, for which we have derived requirements (Sec. I-B). It should be capable of prioritizing test scripts such that a scheduling algorithm can make sure that the more valuable test cases are executed. In addition, it should be capable of dealing with unexpected events such as test system misbehavior. Overall, the assistance system should support human test managers and, thus, interaction and preferences for the system should be possible.

The proposed assistance system can be divided into two phases: static prioritization and dynamic execution ordering. Static prioritization (Sec. III-A) uses Resource Interface Agents to extract relevant information from various sources, which are provided in the form of prioritization factors to the Aggregation Agent. The Aggregation Agent then calculates a static priority based on the provided information by weighing the different factors according to the test managers preferences. Generally, the proposed framework allows to hand-over tasks such as the prioritization factor calculation to autonomous agents. In practice, it remains a challenge to do so as much information is not available or at least not in a machine-readable format. Thus, a User Interface Agent queries a human expert for such inputs. Starting from the static priority, an adaptive execution order is determined (Sec. III-B). The execution order is derived by negotiations between Test Script Agents and Test Bench Agents. Once test benches execute test scripts, the test logs are reported to the respective Test Script Agents, which in turn dynamically adapt their priority values according to a re-prioritization mode chosen by the test manager. As each Test Script Agent represents one Test Script, in the event of special occurrences, related Test Script Agents are informed such that they also dynamically adapt their priorities.

We illustrated the applicability of the proposed framework in Sec. IV in context of HiL-testing in the automotive domain. We do not limit the framework to the automotive context. On the contrary we suggest that it should be equally applicable to other testing contexts as well. However, we also point out that in the presented use case not all information that has been identified as useful for the framework was available. This results in a substantial amount of manual work required to apply the proposed assistance system in terms of information manually provided by the test manager. Nonetheless, the proposed framework has the potential to increase the effectiveness of test management processes by reducing manual work on recurrent tasks such as execution planning. A thorough evaluation in particular of the quantitative effectiveness of the dynamic execution ordering is part of further research work. Therein, it should also be investigated if fuzzy logic is more suitable for deriving static priorities. Moreover, it should be investigated how to deal with unexpected test system misbehavior, which was faced rather frequent in the use case. Thus, a method that automatically recognizes misbehavior and reacts to it appropriately would be highly beneficial.

References

[1] P. J. King and D. G. Copp, "Hardware in the loop for automotive vehicle control systems development," in *UKACC Control 2004 Mini Symposia*, 2004.

[2] S. Abele, A. Zeller, N. Jazdi, and M. Weyrich, "Agentenbasierte Testplanung für industrielle IT-Systeme in der Fertigung," *atp edition*, 2017.

[3] C. Malz, N. Jazdi, and P. Gohner, "Prioritization of test cases using software agents and fuzzy logic," in *IEEE 5th Int. Conf. on Software Testing, Verification and Validation*, 2012.

[4] M. Wooldridge and N. R. Jennings, "Intelligent agents: theory and practice," *The Knowledge Eng. Review*, 1995.

[5] W. Shen, "Distributed manufacturing scheduling using intelligent agents," *IEEE Intelligent Systems*, 2002.

[6] P. K. Arora and R. Bhatia, "Mobile agent-based regression test case generation using model and formal specifications," *IET Software*, 2018.

[7] C. Malz and N. Jazdi, "Agent-based test management for software system test," in *IEEE Int. Conf. on Automation, Quality and Testing Robotics*, 2010.

[8] J. Ahmad and S. Baharom, "Factor determination in prioritizing test cases for event sequences: A systematic literature review," *Journal of Telecom., Electronic and Comp. Eng.*, 2018.

[9] R. Kavitha, V. R. Kavitha, and N. Suresh Kumar, "Requirement based test case prioritization," in *IEEE Int. Conf. on Communication Control and Computing Technologies*, 2010.

[10] A. Di Thommazo, K. Camargo, E. Hernandes, G. Gonçalves, J. Pedro, A. Belgamo, and S. Fabbri, "Using the dependence level among requirements to priorize the regression testing set and characterize the complexity of requirements change," in *17th Int. Conf. on Enterprise Inform. Sys.*, 2015.

[11] F. d. Farzat, "Test case selection method for emergency changes," in *2nd Int. Symp on Search Based Software Eng.*, L. C. Briand, Ed., 2010.

[12] S. Abele and M. Weyrich, "Supporting the regression test of multivariant systems in distributed production scenarios," in *IEEE 21st Int. Conf. on Emerging Technol. and Factory Aut.*, 2016.

[13] S. Vasanthapriyan, J. Tian, D. Zhao, S. Xiong, and J. Xiang, "An Ontology-based Knowledge Management System for Software Testing," in *Knowledge and Sys. Sci.*, 2017.

[14] S. Tahvili, M. Saadatmand, S. Larsson, W. Afzal, M. Bohlin, and D. Sundmark, "Dynamic integration test selection based on test case dependencies," in *IEEE Int. Conf. on Software Testing, Verification and Validation workshops*, 2016.

[15] H. Hemmati, Z. Fang, M. V. Mäntylä, and B. Adams, "Prioritizing manual test cases in rapid release environments," *Software Testing, Verification and Reliability*, 2017.

[16] E. S. Reetz, D. Kuemper, K. Moessner, and R. Tonjes, "Investigation of opportunities for test case selection optimisation based on similarity computation and search-based minimisation algorithms," in *6th Int. Conf. on Adv. in Sys. Testing and Validation Lifecycle*, 2014.

[17] X. Wang, S. Jiang, P. Gao, X. Ju, R. Wang, and Y. Zhang, "Cost-effective testing based fault localization with distance based test-suite reduction," *Sci. China Inform. Sci.*, 2017.

[18] M. Kumar, A. Sharma, and R. Kumar, "Towards multi-faceted test cases optimization," *Journal of Software Eng. and Applications*, 2011.

[19] D. K. Yadav and S. Dutta, "Test case prioritization technique based on early fault detection using fuzzy logic," *3rd Int. Conf. on Comp. for Sustainable Global Development*, 2016.

[20] Y. Yujie, P. Shuwei, L. Huanmin, Z. Xiaoran, and Z. Juan, "Validation test case selection based on multifactor," in *2nd IEEE Int. Conf. on Comp. and Communications*, 2016.

[21] M. Mossige, A. Gotlieb, H. Spieker, H. Meling, and M. Carlsson, "Time-aware test case execution scheduling for cyber-physical systems," in *Principles and practice of constraint programming*, 2017.

[22] Z. Jiang, Y. Jin, M. E, and Q. Li, "Distributed dynamic scheduling for cyber-physical production systems based on a multi-agent system," *IEEE Access*, 2018.

[23] A. Hudaib, M. H. Qasem, and N. Obeid, "Fipa-based semi-centralized protocol for negotiation," in *Cybernetics Approaches in Intelligent Systems*, 2018.