information

Article

The Faceted and Exploratory Search for Test Knowledge

Marco Franke [1,*], Klaus-Dieter Thoben [1,2] and Beate Ehrhardt [3]

[1] Bremer Institut für Produktion und Logistik GmbH, 28359 Bremen, Germany
[2] Faculty of Production Engineering, University of Bremen, 28359 Bremen, Germany
[3] Institute for Mathematical Innovation, University of Bath, Bath BA2 7AY, UK
* Correspondence: fma@biba.uni-bremen.de; Tel.: +49-(0)421-218-50089

Abstract: Heterogeneous test processes concerning test goals and test script languages are an integral part of mechatronic systems development in supply chains. Here, test cases are written in a multitude of different test script languages. The translation between test script languages is possible, a joint understanding and a holistic view of the mechatronic system as a system under test is only achieved in the minds of experienced test engineers. This joined-up information is called test knowledge and is the key input for test automation and in turn, it is essential for reducing the cost of product development. Persisted test knowledge enables the search for patterns semi-automatically without reading countless test cases and enables the auto-completion of essential parts of test cases. In this paper, we developed a knowledge graph that aggregates all the test knowledge automatically and integrates it into the test processes. We derived an explorative search that simplifies the test case creation. For that purpose, a corresponding user-friendly query language, and unidirectional translation capabilities were developed that translates a test case into a graph tailored to the target audience of test engineers. We demonstrated the usage and impact of this approach by evaluating it on test cases from aircraft cabin doors.

Keywords: exploratory search; test knowledge; interoperability of test cases; knowledge graph

Citation: Franke, M.; Thoben, K.-D.; Ehrhardt, B. The Faceted and Exploratory Search for Test Knowledge. *Information* **2023**, *14*, 45. https://doi.org/10.3390/info14010045

Academic Editors: Diego Reforgiato Recupero, Pierpaolo Basile and Annalina Caputo

Received: 30 September 2022
Revised: 8 January 2023
Accepted: 9 January 2023
Published: 11 January 2023

1. Introduction

Mechatronics is a multidisciplinary branch of engineering integrating electronics, control engineering, mechanical engineering and, in some cases, microprocessors [1,2]. It focuses on the development of a compound that combines multiple distributed, discrete-event systems that together fulfil a single specific purpose [3]. Examples of such mechatronic systems are cars, aircraft, and trains. The development of these products is performed in a supply chain where currently, a test process at the original equipment manufacturer (OEM) and several test processes by the suppliers are carried out separately. The aim of the test process is the verification of the mechatronic system with test cases proving the correct behaviour of the system. In the case of aircraft, the related verification effort represents a significant portion of avionics development costs. For example, the verification of software in avionics made up 40% of the development costs in 2001 [4]. The amount of software testing has been increasing on account of technological developments [5,6]. This increasing complexity must be managed by the test processes in the supply chain.

Within a supply chain, there are multiple test processes which are heterogeneous regarding their goals, the applied test script languages, and test benches that are used. A side effect of the development process is that all specified test cases define the complete functionality and behaviour of the aircraft from the perspective system under test (SUT). This is a source of knowledge that is called 'test knowledge'. Up to now, the different parts of the test knowledge were analysed in isolation and set apart by the varying test script languages. The challenge we set here was to aggregate and then use the test knowledge over the boundaries of test script languages and test processes.

For many supply chains, an exchange of test knowledge between test processes is necessary to compare the test case semantics, even though they are often represented in different test script languages.

Currently, an interoperable representation of test cases is created using state charts [6,7] which translate test cases between test script languages. This approach enables the direct exchange of test cases without the need to understand their content. The main shortcoming is that the interoperability of test cases is only used for source-to-source compiling of test cases and not to create a holistic view of the aircraft as a SUT. This test knowledge is only available in the minds of experienced test engineers. However, it is crucial as input for test automation [8–10]. A significant effort is spent in the creation of test cases whereby each test case proves the correct functionality of a system function. An essential part of a test case is the proper stimulation of the unit under test (UUT). It requires not only knowing which signals are necessary for the stimulation of the UUT but also how the entire system under test (SUT) needs to be stimulated. Without the correct stimulation of the SUT, testing the UUT is impossible. In these addressed cases, the SUT can be, for example, the high-lift system of an aeroplane or the cabin of an aeroplane. Accordingly, test case creation requires profound system knowledge, which is obtained manually through specifications, system requirements, and test cases. The number of test cases ranges from a couple of hundred to many thousands, depending on the system. To achieve a more direct exchange of test knowledge, search methods for the correct stimulation of the UUT/SUT are necessary; thus, it is the key to countering the increase in test efforts.

The solution presented here enabled us to create a holistic view for the first time automatically. We developed a knowledge graph taking the heterogeneous test cases as input and gain test process-related search capabilities.

Knowledge graphs were introduced as "Knowledge Graph, which will help you discover new information quickly and easily" by Google in 2012 [11], which was established before in the scope of the Semantic Web. In doing so, the evolution of the Internet enables the exchange of information instead of data. Information-driven services are in place to offer added value. A central tool for modelling information for the purpose of information modelling and information integration are ontologies [12]. In the scope of the Semantic Web, ontologies have been heavily used to reinvent information provision. For that purpose, a wide range of top-level and domain ontologies were developed. The original intention of knowledge graphs, as well as the intention of this article, was to improve search capabilities and enhance user experience. Soon after, high-quality, large-scale knowledge graphs were developed. For example, DBpedia represents extracted content by Wikipedia [13].

The technology knowledge graph is flexible enough to map test cases and their contained test knowledge into a graph. The approach presented here achieves the vision of interoperability, defined as "the ability of two or more systems or components to exchange and use information" [14]. For that purpose, the authors focused on a paradigm change: The content of the test cases shall be handled not as data but as information. For that purpose, a suitable information model was needed, enabling the aggregation and provision of test knowledge. Building on that, we delivered a facet and explorative search of test knowledge in the supply chain's test processes, enabling test case-relevant search capabilities.

The remainder of this paper is structured as follows. Section 2 describes the conceptual foundations of this article and the related work. Section 3 presents the concept, followed by the description of one application scenario in Section 4. In Section 5, a short discussion is given. Finally, Section 6 presents the conclusion. This work is based on preliminary work related to the translation from test cases to Generic SCXML. The preliminary solution is customised and applied in Step 1 whereby the approach has been introduced in [6,7]. In addition, the state-of-the-art according to test script language is taken from preliminary work [15].

2. Background and Related Work

2.1. Article Relevant Definition

The article uses terms from the domains of testing and software development, which are briefly introduced for this article. The relevant definitions are shown in Table 1.

Table 1. Definitions.

Definition	Description
Block	A block is a structural element used to group statements within a test case. Most of the time, a block describes a test step in the test case.
State	A state describes the current state of a system in a state diagram or in test execution. An example of a state is the off state when all of the aircraft's systems are off. Hereby a state can be divided into two parts: micro-state and macro-state. The micro-state defines all statements which shall be executed immediately. The macro-state describes the result of the execution of the statements.
Statement	A statement is a line within the source code of a test case that executes instructions. For example, assigning a number to a variable is a statement.
Stimulus	A stimulus is a statement that leads to a change of state. For example, the stimulus to turn on the light causes the lamps to glow afterwards
Ramp	A ramp is a repetitive stimulus in which a signal is increased at a predefined slope.

2.2. Knowledge Graph in the Scope of Test Knowledge

There are a couple of different definitions of knowledge graphs [16–18]. We apply the definition by Hogan [19]: "Herein, we define a knowledge graph as a graph of data intended to accumulate and convey knowledge of the real world, whose nodes represent entities of interest and whose edges represent potentially different relations between these entities.". The developed graph for representing test cases defines the SUT and is described by a varying number of test cases (i.e., nodes). It contains information about the stimuli and correct behaviours defined for a specific requirement or functional aspect. This test knowledge can be represented differently depending on the applied model. To be more precise, the knowledge graph is defined as a Directed Edge-labelled Graph [16]. Each node represents a real object, and the directed, labelled edges define the relations between states and the properties of the edge. Each node and label can have a type to include taxonomies as introduced in Heterogeneous Graphs [16]. If a real object needs to be described via a property, the Property Graph [16] is applied.

Graphs can be modelled to describe or to visualise ontologies. They are commonly used to model information independently from a specific data format. As discussed in artificial intelligence (AI), ontologies are formal, partial specifications of an agreement over the description of a domain [12], and consist of concepts and relations and their definitions, properties, and constraints expressed by axioms [20]. Ontologies have been applied as part of the Semantic Web. In doing so, ontologies exist on different levels, each with a specific purpose. There are aims to cover basic concepts and properties in so-called top-level/upper ontologies. Examples of top-level ontologies are e.g., basic formal ontology (BFO) or descriptive ontology for linguistic or cognitive engineering (DOLCE).

On the contrary, domain ontology describes a specific domain and domain ontologies are available in uncountable domains. In the scope of testing, there are no ontologies describing the content of test cases. The domain ontologies for testing, such as STOWS [21] or SWTO [22], describe the structure and the proceeding of a test process and not the content of the test relevant documents such as requirements or test cases. There are approaches to represent ontologies as a combination of Heterogeneous Graphs and Property Graph using Neo4J as a graph database.

This article focuses on mapping test cases and their contained test knowledge into a graph. For that purpose, the goal, the structure, the heterogeneity, and the information to be homogenised are defined by test cases and their integration into test processes for

the development of complex mechatronic systems. Querying information on graphs is a mandatory feature to enable the development of information-driven services. The querying capabilities depend on the underlying graph type. The common approach is to provide a query language such as SPARQL, Cipher or Gremlin to query data. However, commercial solutions are available for each graph type and query language that are improved continuously and are outside the scope of this work.

2.3. Searchable Information inside a Test Case

The intended usage of knowledge graphs is to store the content of test cases and share the contained test knowledge. Shareable test knowledge is, for example, the correct stimulation of a unit under test (UUT), considering all dependencies to the system under test (SUT). Moreover, the correct stimulation always needs to know how quickly the system can be stimulated. This knowledge is acquired through trial and error in each new test process and is therefore valuable for the exchange. Up to now, this test knowledge has been gained manually by reading the system requirement document (SRD) and gained by test cases from previous test campaigns. Based on the information sources, the test engineer develops a basic understanding of the system and completes these by several failed tries of stimulating the SUT. Due to the increasing complexity of aircraft, while relying on similar system functions across the boundaries of individual product generations, there is the possibility to aggregate test cases into a knowledge graph and offer the included test knowledge.

The prerequisite of the motivated search capabilities is the interoperability of test cases. The aim of interoperability is the capability to model the content of a test case independently of the selected test script language and to integrate its content into another test process. In doing so, the integration means translating the content of the original test case into a graph. The interoperability of test cases shall focus on the aggregation to test knowledge that is not possible yet, because test script languages are heterogenic. In the following, the heterogeneity of test script languages is presented in detail. With this, the shown heterogeneity needs to be harmonised to achieve interoperable test cases and enable search capabilities.

The available test script languages have so far been based on imperative programming. As a result, a test case consists of a set of statements that are executed sequentially. Although test cases have a similar structure, the test script languages are heterogeneous in terms of their syntax, semantics and provided functions. In the following section, the structure of a minimalist test case is first described. Based on the minimalist test case, the reasons for the different implementations of the same test case structure in the test script languages are explained.

2.3.1. Structure of a Test Case

The following state of the art is preliminary work and has already been published [15]. The repetition is necessary to understand test case structure, content, and representation, which is the baseline for the developed graph G in step 2, presented in Section 3.2.

A test case consists of a sequence of statements. The sequence aims to prove that the SUT works according to the requirements. In the scope of testing mechatronic systems, it means that the specified test data, the expected results and a description of the steps to be tested are specified and mapped as a sequence of statements [23]. International Software Testing Qualifications Board (ISTQB) [24] and the IEEE 829 [14] give a congruent definition of the test case. According to this definition, a test case includes the following information: the preconditions necessary for the execution, the set of input values (one input value per parameter of the test object), the set of predicted results, and the expected post conditions. Test cases are developed concerning a specific goal or test condition, such as executing a specific program path or checking compliance with specific requirements [24].

Both definitions introduce the specified test data and the expected results as components of a test case. The test data are fed in various ways and stimulate the SUT. Based

on the stimulation, results are expected on how the SUT must behave according to the requirement. The expected results are defined as a condition, and the condition check is implemented using statements of the type of *verdict*. Examples of verdicts are the statement *expect* in Check Case Definition Language (CCDL) (https://www.razorcat.com/files/de/produkte/ccdl/Razorcat_Technical_Report_CCDL_Whitepaper_02.pdf, accessed on 5 January 2023) or *result* in Real-Time Testing (RTT) (https://www.dspace.com/de/gmb/home/products/sw/test_automation_software/automated_test_execution_in_re.cfm, accessed on 5 January 2023).

A mechatronic product is a network of distributed, event-discrete systems [1], which is an information system that implements its control function using software [3]. Correspondingly, test cases for testing mechatronic systems as information systems contain statements for stimulating the SUT as well as verdicts. As part of a mechatronic product, a mechatronic system behaves like a black box. Thus, it can only be stimulated via its interfaces. In addition, the conditions for the verdicts can only be defined via signals that are available via the interface. The integration of software and actuators means that the stimulation of a signal is not executed immediately but requires time, depending on the system. This time behaviour must be taken into account in the test cases. Statements for controlling the runtime behaviour are provided for this.

The structure of a minimalist test case for testing safety-critical control functions is shown in Figure 1. It contains the statements mentioned above and is defined for testing mechatronic systems by [7]. In other common test cases, the failure state would be replaced by a target state to check whether the desired function works under predefined conditions.

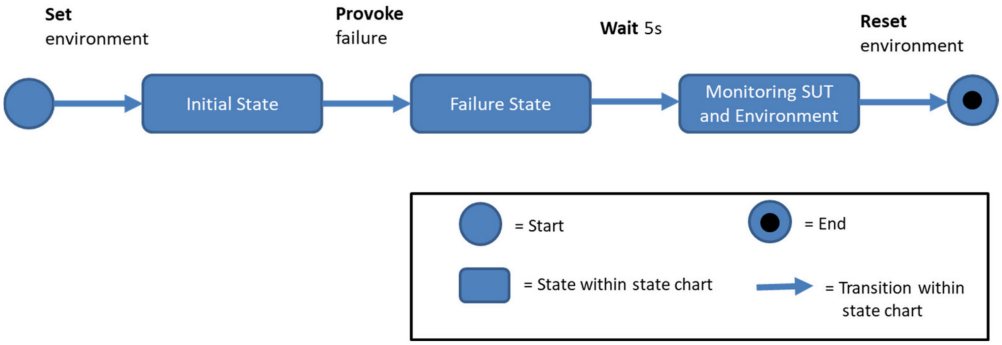

Figure 1. Basic steps of a test case.

As shown in Figure 1, a test case stimulates three consecutive states in the SUT. First, the test case stimulates the environment [25]. The error is then stimulated, and the SUT is finally observed. This proceeding of states is achieved by executing a sequence of statements. The state-changing statements are attached to the arrows in the figure. In order to check whether the SUT has reached a target state or whether it has waited long enough, statements are listed within the state.

In addition to states and transitions, a real test case also includes test steps and test bench-specific functions [6]. The minimalistic test case already contains different types of statements that perform different tasks. Figure 2 shows the mapping of the basic structure of a minimalist test case to the statement types of the test script languages.

It can be seen here that statements of the stimulus type are used to achieve the initial state of the SUT, the error state and to reset the environment. The waiting within test case execution is needed to wait for the reaction of the SUT. For that purpose, statements for controlling the timing are used. Finally, the system's response is evaluated using statements of the type of verdicts.

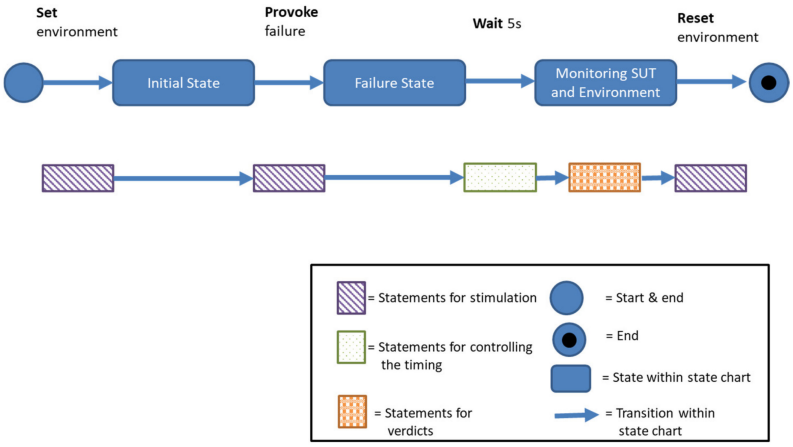

Figure 2. Mapping of test case steps to statement types.

The statement types listed above (statements for stimulation, statements for controlling the timing, statements for verdicts) are necessary for providing the basic functionality of a test case and are available in all test script languages. The most common languages among them are for testing mechatronic systems (such as cars and aeroplanes): ASAM XIL (https://www.asam.net/standards/detail/xil/, accessed on 5 January 2023), Testing and Test Control Notation version 3 (TTCN-3) (http://www.ttcn-3.org/, accessed on 5 January 2023), CCDL, C (https://en.wikipedia.org/wiki/C_(programming_language), accessed on 5 January 2023) or RTT, and Python (https://www.python.org/, accessed on 5 January 2023). The following differences between these languages are presented in detail to describe the problem statement.

2.3.2. Structure of Test Script Languages

The available test script languages are imperative programming languages and can be divided into the groups "safe" and "unsafe" [25]. A "safe" language prevents the language from being misused. For this purpose, additional mechanisms such as type security of the variables or unique access methods for data structures are added in order to avoid inconsistency in the internal representation of the program execution. An example of such an inconsistency would be when the beginning of two arrays in memory have overlapping memory addresses and affect each other. In contrast, "unsafe" languages do not have these restrictions. The programs can be checked statically (on compilation) as well as dynamically with respect to the variables and their types at runtime (on runtime).

Apart from the two groups of test script languages, the typing, the programming paradigm, and the execution modus have an essential impact on the syntax of a test script language. Examples of the impact of the programming paradigm and type safety are shown for CCDL, JAVA (https://www.java.com/en/, accessed on 5 January 2023) and Python in Figure 3.

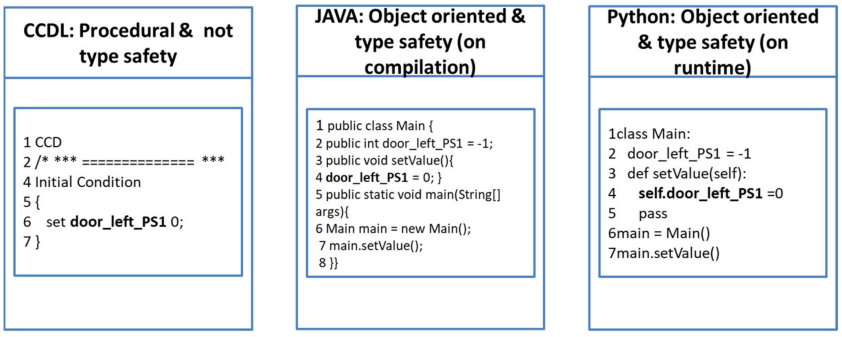

Figure 3. A similar test case in different test cases languages.

All three examples set the signal door_left_PS1 to the value 0, while the test cases' syntax differs. The interoperability of two test cases from different test script languages must overcome the presented heterogeneity related to typing, programming paradigm, and execution modules. The benefit of the interoperability would be to gain two test cases that can be executed on the supplier's test bench or to aggregate the test cases within a knowledge graph.

2.3.3. Heterogeneity in Language Complexity of Test Script Languages

The different test script languages are heterogeneous according to their range of functions, which is reflected in their syntax. In the following, the heterogeneity is shown for the example: ramp statement, which is an adapted version of a loop statement to stimulate a signal over a specific duration.

The heterogeneity with respect to the semantics of the function ramp for the test script languages CCDL and ASAM XIL API (https://www.asam.net/standards/detail/xil/older/, accessed on 5 January 2023) is shown in Figure 4 below. In doing so, the figure shows an intersection known from the set theory whereby the set elements are the input parameters of a ramp. These two test script languages were selected as examples because CCDL is primarily used in aviation, and ASAM XIL API is primarily used in automotive applications.

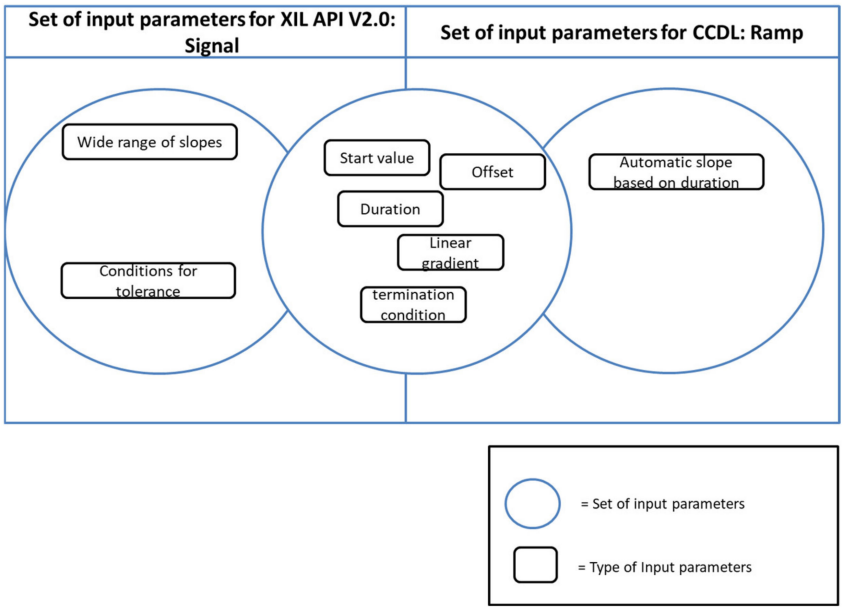

Figure 4. The intersection of the input parameters for the ramp function in XIL and CCDL.

The first difference is the name of the statement. While the ramp statement is called signal in XIL API, it is called ramp in CCDL. Different names for similar test process-specific statements make it difficult to compare different test cases from different languages without being an expert in each test script language. Apart from the name of the function, the implementation in XIL and CCDL offers a similar subset of the functionality. Apart from this, the language offers domain-specific features. In the case of CCDL, the ramp can be implemented without any predefined slope. Without further transformations, this ramp variant cannot be mapped to the XIL signal statement.

The example of a ramp statement, which is transferable to all other test process-specific statements, shall demonstrate the heterogeneity. This heterogeneity needs to be harmonised to gain applicable interoperable test cases.

2.4. Summary

In general, a knowledge graph is suitable to represent knowledge for a domain, which is shown by the availability of upper and domain ontologies. The choice of the graph type and query language determines which content can be stored and queried.

The targeted domain for aggregating, storing and querying test knowledge is the test process in a supply chain. For this purpose, test cases need to be transferred into the knowledge graph. The challenge here is that the test cases are heterogeneous in terms of their syntax and semantics. Accordingly, a 1:1 transfer, i.e., a transfer of the sequence of instructions, would also transfer the heterogeneity related to syntax and semantics of the specific test script language. The result would be uploaded test cases in the knowledge graph that would be similar according to their semantics but would have a different sequence of nodes and edges from the graph's perspective. The resulting heterogeneity would prevent a suitable explorative search approach, which could be defined over more than one test case. Thus, the aggregation of test cases requires complex pre-processing and a graph scheme that focuses on the functionality of the test case and abstracts from the actual syntax.

3. Concept of a Knowledge Graph for Test Knowledge

The goal is to store test knowledge in a knowledge graph and the subsequent provision of the faceted and explorative search in the supply chain's test processes. The developed method is shown in Figure 5, which consists of three sequential steps, namely Step 1: Translation of the test case into a test model, Step 2: Import test models into a knowledge base and Step 3: Provide test-relevant services. The method's input is test cases implemented in different test script languages, and the output is information offered by two search modes. The faceted search enables the test engineer to narrow down search results related to SUT signals and their value ranges. The explorative search takes a pattern as input and offers the auto-completion of test cases.

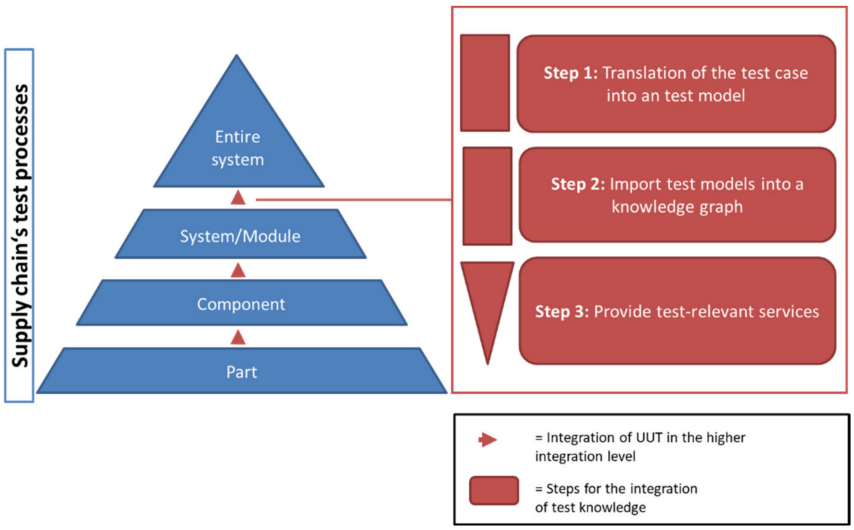

Figure 5. Extraction and integration of test knowledge into the supply chain test processes.

The developed method transforms the data into knowledge. The transformation process follows the layers of the DIK pyramid [26]. With this, the data are the content of different test cases, the test model (output of step 1) is the interoperable representation of the test cases, and the knowledge graph (output of step 2) is the search-optimised representation of the test model. At this moment, information is "data that has been given meaning by way of relational connection" [27]. The relational connections describe the semantics of a test case and how a test engineer understands the content of a test case. In the following section, the three steps are presented in detail.

3.1. Step 1: Translation of the Test Case into a Test Model

The goal of the first step is to transform the content of test cases from different test script languages into a test model, whereby the test model fulfils the property of interoperability on the data/information level [28]. Despite a test case being already semi-structured and the semantics being precise, the content of the test case is handled as data, transformed into information, and stored in an interoperability model. The advantage is that the information on how a SUT can be stimulated and how the SUT behaviour can be checked is captured, which is highly relevant to the test engineer (see Section 2.3). Other semantics resulting from the structure of the test script language used are not relevant to the semantics of the test case. For example, the taxonomy of the classes and objects within a test case from an object-oriented test script language is irrelevant to the stimulation of the SUT and will be removed during information extraction. The reduction of the test case to its semantics only contains the information on how the SUT is stimulated, how the SUT behaviour is then checked and how the timing of the test case was planned. This makes a test case no longer executable on an execution engine, but it makes it comparable across all test script languages. Such an approach was first developed by Rasche et al. [6,7]; it is the preliminary work of these authors and is extended here to the latest version of Generic SCXML [29]. The representation of a test case in SCXML differs significantly between the preliminary work by Rasche et al. [6,7] and the latest version of Generic SCXML. Major differences are:

- The latest version of Generic SCXML introduced TASCXML commands to replace the test automation function represented natively in SCXML states, including its complete semantics;
- A test case's overall structure is not compatible any longer with the outcome by Rasche et al.;
- The meta-information has been completely revised and is not compatible with the outcome by Rasche et al.

With this, the translation of a test case is similar to source-to-source compiling. While the syntactic processing is based on parse trees, the semantic processing was specifically processed in the direction of a state-based perspective. The translation process from a test case to an SCXML model is shown in Figure 6:

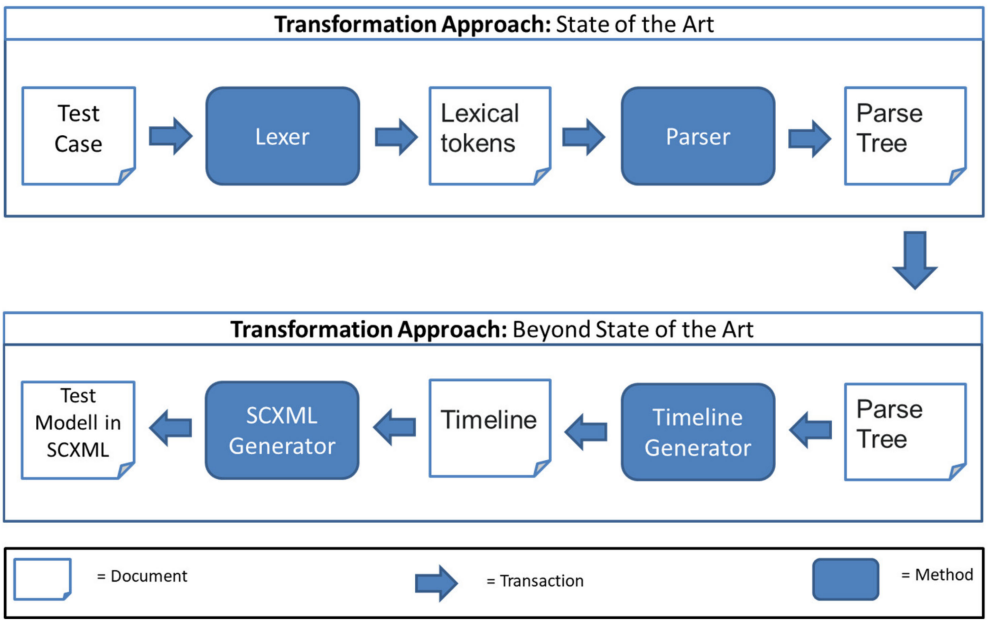

Figure 6. Transformation Approach, adapted from [15].

For each test script language, a specific lexer and parser are required. The definition and creation of such tools are state of the art and work similarly for all test script languages. The result of the execution of a parser is a parse tree, which models the test case structure as a tree. This tree depends on the properties of the test script language according to, e.g., the programming pattern or the type safety (see Section 2.3).

The harmonisation step to achieve interoperability is located in the timeline generator. The timeline generator is specific for a test script language and generates from the parse tree a sequence of statements. Hereby, the statement sequence is a timeline that defines when each statement should be triggered during a test case execution. The timeline does not define how long the statement runs, but rather, it defines whether the statement is blocking or non-blocking. Based on the timeline, the SCXML Generator first creates the overall test case structure (StartUp, Logic, TearDown). This test case structure has been developed as the typical structure for testing aircraft in the research project AGILE-VT [29]. Subsequently, the timeline generator adds all statements chronologically. In doing so, blocking statements are added as actions inside the predefined states. Parallel statements can be added by creating a parallel state which would be the first child of the state logic. The outcome of step 1 is a set of test cases from different test script languages in SCXML and, more precisely, in the last Generic SCXML version.

3.2. Step 2: Import Test Models into a Knowledge Graph

The outcome of the first step is a set of test cases in SCXML, which are state charts and are independent of their original test script languages. Therefore, interoperability on the data/information layer is achieved for test cases. Thus, the prerequisite for aggregating test cases in a knowledge graph is basically fulfilled. The aggregation needs a repository and a database from a technical point of view. With this, a test case in SCXML can be uploaded as a large binary object (BLOB) to structured query language (SQL) databases or natively as a parseable extensible markup language (XML) document into a NoSQL database. The weakness of a BLOB is that it works as a black box, and no information can be queried using the database's query language. Thus, an SQL database is no appropriate solution. The query language of a NoSQL database focussing on XML uses XML Path Language (XPath) or XQuery to some extent as query language. The drawbacks of using XPath (The findings also apply to XQuery) for implementing a faceted and explorative search are presented in the following. In doing so, XPath defines search queries over the XML tree structure. The weakness of using the tree structure for querying specific elements in test cases, which have the same semantics and are represented in SCXML, is that test cases have varying tree structures. This problem is described in the following section, based on an abstract test case in SCXML.

All test cases in an NoSQL database would have the same hierarchical structure defined over this sequence of compound states: StartUp, Logic, and TearDown. An example of the test case structure is given in Figure 7.

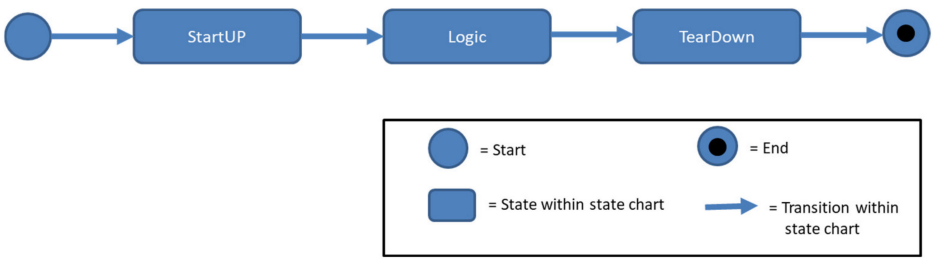

Figure 7. Basic Structure of a test case in SCXML.

This structure already allows for defining the same query for the top-level compound states in all test cases, which is necessary to create facets and find generic patterns. In the XPath notation, an example of querying the StartUP in all test cases would be "scxml/state[@id ='StartUP']". Within each of the compound states, there is no design pat-

tern specified. The implication is that the sequence of XML elements within the compound states differs between test cases. It means that the test engineer defines, for example, how many states he would like to use for the stimulation of the aircraft. Figure 8 demonstrates two possible design patterns of how the stimulation could look in an SCXML test case that follows the naming convention of Generic SCXML.

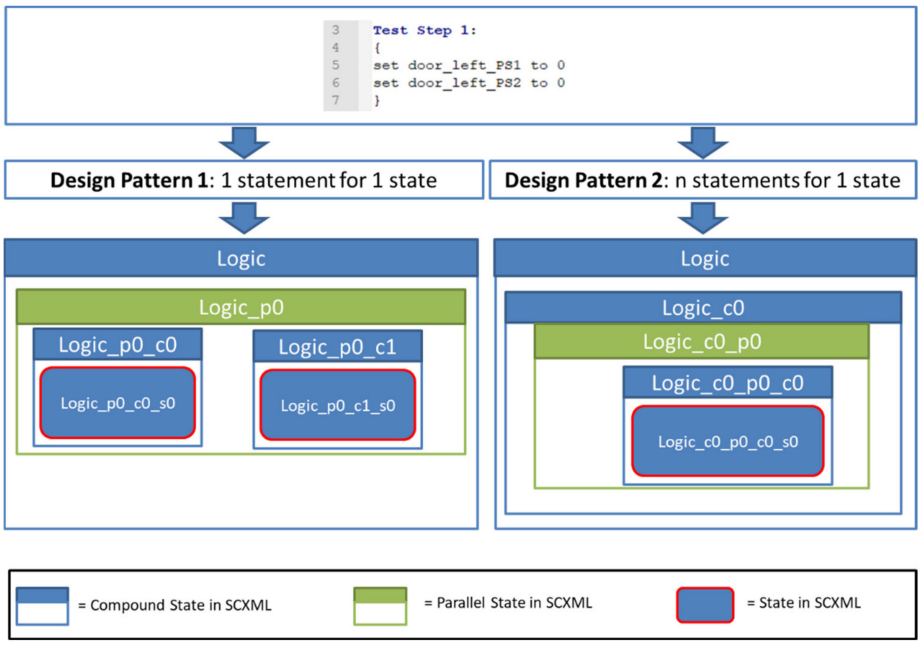

Figure 8. Example of different design patterns for the same test case in CCDL.

Design pattern 1 is that only one signal is stimulated with one new value in each state. Thus, the example for the design pattern 1 has two states {Logic_p0_c0_s0, Logic_p0_c1_s0}.

On the contrary, design pattern 2 recommends adding all statements to one state. Thus, the example for design pattern 2 has one state {Logic_c0_p0_c0_s0}. Both design patterns result in a completely different tree structure. More design patterns are available and increase the structural heterogeneity. It is not a problem in the scope of the source-to-source compiling but for implementing the intended querying method of step 3. For that purpose, an additional interoperability challenge must be solved.

The states' taxonomy is an XML tree, and it is the baseline of how XPath can query information of an SCXML test case. To request specific information, XPath needs to know the states' taxonomy beforehand to define the correct query. The example above was "scxml/state[@id =' StartUP']". The definition of all possible queries can be defined based on grammar. Thus, the XPath query for an SCXML test case is always defined by the following grammar, whereby the SCXML statement specific part is shortened to the string statement:

grammar XPathQuery; (1)

query: (root compoundstates body*) | (SEPARATOR SEPARATOR body); (2)

root: SEPARATOR ROOT; (3)

ROOT: 'scxml'; (4)

compoundstates: SEPARATOR 'state [@id=\" ID '\']'; (5)

ID: 'StartUP' | 'Logic' | 'TearDown'; (6)

body: ((SEPARATOR'state [@id=\" NAME '\'] ') | (SEPARATOR 'parallel [@id=\" NAME'\'] '))+ '/onentry' 'state-ment'; (7)

NAME: [a-zA-Z0-9'_']+; (8)

SEPARATOR : '/'; (9)

The grammar shows that the application of the nonterminal symbol (2, 4, 5) supports the definition of the basic structure of a testcase in SCXML (see Figure 7) as an XPath query. For example, the query */scxml/state[@id ='StartUP']* can be constructed and already contains an XML tree of deep 2. The nonterminal symbol (7) allows the iteration through all deeper elements of the XML element tree inside an SCXML test case. An XPath query for the design pattern1 from Figure 8 would be */scxml/state[@id ='Logic']/parallel[@id ='Logic_p0']/state[@id ='Logic_p0_c0']/state[@id ='Logic_p0_c0_s0']*. Thus, the element tree and the XML attribute ID are mandatory parts of each search query. These elements are test case specific; therefore, a query cannot find results from more than a test case. There is one exception, which is the XPath operator //. It is defined as searching everywhere in the XML tree structure. It would work for facets but not for the intended explorative search. If the user would like to search for the stimulation of two signals, the state context would be needed starting from the second statement. Consequently, aggregating test cases as SCXML models is not an appreciated solution.

The paper aims to create a knowledge graph that includes all test cases and supports an explorative search. The search enables querying of stimulation, conditions, timing, etc., in all test cases. This means that the knowledge graph shall focus on the test procedure, including its functional point of view of a test case, instead of its structure. For that purpose, different graph types can offer nodes, edges, labels, and properties. In the following section, the representation of a test case as a graph is presented in detail.

3.2.1. Nodes and Labels

The research project AGILE VT [23] has identified the relevant set of test-specific statements as TASCXML commands, shown in Table 2. As a result, the command set shall enable the implementation of Hardware in the Loop (HIL) test cases in the automotive and avionics domain that cover most of the basic statement types shown in Figure 2.

Table 2. TASCXML commands.

Id	TASCXML Commando	Type of Statement	Description
1	<TASCXML:set>	Stimulation	Sets a signal to a specific value
2	<TASCXML:get>	Stimulation	Reads the current value of a signal
3	<TASCXML:result>	Verdicts	Logs a result (verdict) of the test case
4	<TASCXML:ramp>	Stimulation	Triggers a ramp that changes a signal for a specified duration
5	<TASCXML:sine>	Stimulation	Generates a curve shape of a sine curve
6	<TASCXML:sawtooth>	Stimulation	Creates a waveform of a sawtooth
7	<TASCXML:pulse>	Stimulation	Creates a waveform of a pulse
8	<TASCXML:verifytolerance>	Verdicts	Verifies whether the value of a signal is within a specific tolerance range

Each TASCXML command except for <TASCXML:get> shall be represented as a graph node. The distinction between two TASCXML statements is implemented using labels. For each type of TASCXML command, its own label is defined. Thus, the proposed graph implements seven labels, namely {'*SET*', '*RESULT*','*RAMP*', '*SINE*','*SAWTOOTH*', '*PULSE*', '*VERIFYTOLERANCE*'}.

Apart from the TASCXML command set, a typical test case defines which statements are executed in parallel, sequential and within a predefined state. To enable the grouping of TASCXML commands to a given state, an additional graph node with the label {state} is defined.

3.2.2. Edges and Labels

Edges connect two nodes in a graph. In the scope of a test case, an edge connects two nodes whereby the node represents a state or a TASCXML command. The edge defines

the timing between the nodes. For that purpose, an edge has a label which describes whether the execution of this node is executed in parallel, sequential or should be delayed. In the latter case, the property defining the duration is needed. The usage of this restricted set of labels shall simplify and harmonise the structure of test cases as part of the graph. In the following, the complete definition of graph G is given.

3.2.3. Graph Definition for Test Cases

We model the test case as a directed graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ in which nodes $v_{1..i}$ in $\mathbf{V}$ denotes a state within the test case, i.e., a state of test execution. A state within the test case execution has two parts. The first part covers the execution of the statements at the beginning of the state as micro-states and the following part covers the result of the stimulation with respect to the SUT as a macro-state. With this, a node can represent one micro-state or one macro-state but not both. That means two connected nodes at least define a SCXML state, which contains a TASCXML command and the resulting SUT state. The separation between the micro and macro-states is implemented with node labels. Hereby, a set of micro-states, which are TASCXML commands, are always connected to a macro-state. The semantic of a macro-state is the representation of a new SUT state in the scope of a test case execution. For example, the micro-state command *to close the door* and the result of the command's execution are already available in the macro-state. The needed information for a micro and macro-state is represented inside the node. Each node v_i is an ordered pair $(\{p_i\}, l_i)$ of a set of properties in P and a label l_i in $L1$ where

$$L1 = \begin{Bmatrix} 'SET', 'RESULT', 'RAMP', 'SINE', 'SAWTOOTH', 'PULSE', 'VERIFYTOLERANCE' \\ 'STATE' \end{Bmatrix} \quad (10)$$

Nodes with the label '$STATE$' define a macro-state in which all connected TASCXML commands shall be executed simultaneously. In doing so, the supported TASCXML commands are defined by the labels as {'SET','RESULT','RAMP','SINE','SAWTOOTH','PULSE', 'VERIFYTOLERANCE'}. One property $p_i = (n_i, z_i)$ is themselves a pair with a string n_i assigning a name to the property as key and z_i capturing the value of the property. The type of the value covers the primitive data types that can be saved as string, number or Boolean. The set of properties as P = {$p_0 \ldots p_n$} defines the set of all relevant properties for a node, respectively. An example of properties for the TASCXML command TASCXML:set are p_0 = {"variable", "string"} and p_1 = {"value", "number"}. With these properties, the command TASCXML:set know which value shall be assigned to the signal. A concrete example is given in Figure 9.

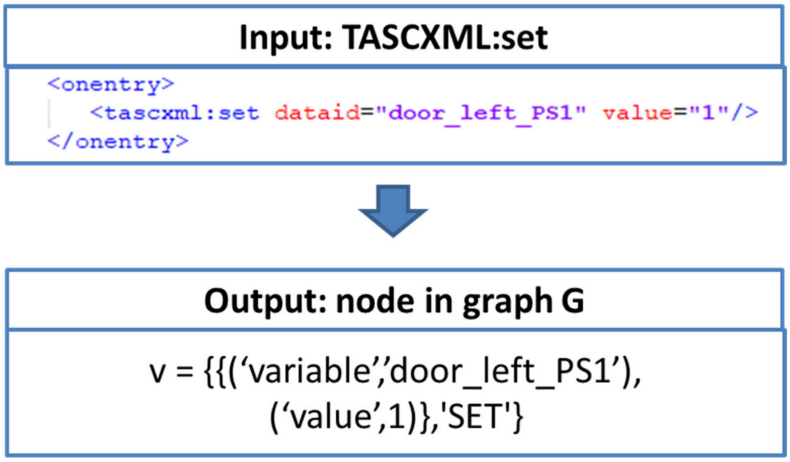

Figure 9. Example for mapping the TASCXML command set to a graph node.

There is an edge e in **E** from node v_1 to node v_2 either a) if there is a transition between two defined states of test execution or b) if a TASCXML command is executed within the same state of test execution. Each edge e in **E** comprises the start node v_1, the end node v_2, a label l_j and a property p_j specifying the length of duration, i.e., $\mathbf{E} = (\mathbf{V}, \mathbf{V}, L2, P2)$. With Z being integers, $L2$ and $P2$ are defined as

$$L2 = \{'SIMPLE', \text{'PARRALLEL\_STATEMENT'}, 'BLOCKING\_STATEMENT', 'wait'\} \tag{11}$$

$$P2 = \{('duration', Z)\} \tag{12}$$

3.2.4. Mapping of a Test Case in SCXML to Graph G

The defined graph G and the SCXML model from the first step represent test cases in different formats and for different purposes. While a test case in SCXML is applied for the source-to-source compiling, the developed graph G shall be used for aggregation of test knowledge based on it to offer search and auto-completion functionality. With this, only a subset of the SCXML test case content is required and defined as graph G.

The mapping of a test case from SCXML schema and its extension to Generic SCXML [29] into a graph G is possible. For that purpose, traversing an SCXML test case and the execution of two functions, namely **f** and **f2**, are required. In the following, both functions are presented in detail.

$$\begin{gathered}\mathbf{f} \text{ is a non} - \text{injective function.} \\ \mathbf{f}: \text{ x} - > \text{v where x is an XML element of SCXML model and v is in } \mathbf{V}.\end{gathered} \tag{13}$$

The variable x is defined as a snippet of the Generic SCXML XML schema definition (XSD). It defines all possible SCXML models for a specific mapping to a specific v. The snippet is always the definition of an XML element that contains the TASCXML command within a Generic SCXML test case. Moreover, v defines the concrete node to a given TASCXML command.

In the following section, the mapping from x to v is defined in Table 3, whereby the mapping focuses on the XML elements as input in the traversing of the SCXML test case and V from the graph. In doing so, all attributes of an XML element are mapped 1:1 as properties P in the graph node V. A detailed listing of all attributes for each XML element is not given.

Table 3. Mapping of an SCXML model to Graph G.

x	**v**
<xsd:element name="state" type="scxml.state.type"/>	v = {P,'STATE'}
<xsd:element name="set" type="scxml.state.type"/>	v = {P,'SET'}
<xsd:element name="result" type="scxml.state.type"/>	v = {P,'RESULT'}
<xsd:element name="ramp" type="scxml.state.type"/>	v = {P,'RAMP'}
<xsd:element name="sine" type="scxml.state.type"/>	v = {P,'SINE'}
<xsd:element name="sawtooth" type="scxml.state.type"/>	v = {P, 'SAWTOOTH'}
<xsd:element name="pulse" type="scxml.state.type"/>	v = {P, 'PULSE'}
<xsd:element name="verifytolerance" type="scxml.state.type"/>	v = {P, 'VERIFYTOLERANCE'}

An example of mapping an SCXML element to graph node v is shown Figure 9.

The traversing of the SCXML test case needs not only a mapping of XML elements to V of the graph but also a mapping of XML elements to E of a graph. Hereby, the mapping is not defined from one XML element to a graph node; instead, a pattern described via

a couple of XML elements is mapped to an E of the graph. For that purpose, there is the non-injective function f2 required which is presented in the following in detail.

$$\mathbf{f2}: \text{x} -> \text{e where x is a set of XML elements of SCXML model and e is in } E. \quad (14)$$

In the following, the mapping from x to e is defined in Table 4.

Table 4. Mapping of SCXML nodes to graph edges.

x	e

`<xsd:group ref="scxml.state.mix" minOccurs="0" max-Occurs="unbounded"/>`
`<xsd:group name="scxml.state.mix">`
`<xsd:choice>`
`<xsd:element ref="state" minOccurs="1" maxOccurs="unbounded"/>`
`</xsd:choice>`
`</xsd:group>`
	e = ((P,'STATE'), (P,'STATE'), 'SIMPLE', P)

`<xsd:group ref="scxml.state.mix" minOccurs="0" maxOccurs="unbounded"/>`
`<xsd:group name="scxml.state.mix">`
`<xsd:choice>`
`<xsd:element ref="onentry" minOccurs="1" maxOccurs="1"/>`
`</xsd:choice>`
`</xsd:group>`
	e = ((P,'STATE'), (P, l ϵ {'SET','RESULT','RAMP','SINE','SAWTOOTH','PULSE','VERIFY-TOLERANCE'}), 'PARRALLEL_STATEMENT', P)

`<xsd:complexType name="scxml.onentry.type">`
`<xsd:group ref="scxml.onentry.content"/>`
`<xsd:attributeGroup ref="scxml.onentry.attlist"/>`
`</xsd:complexType>`
`<xsd:group name="scxml.onentry.content">`
`<xsd:sequence>`
`<xsd:group ref="scxml.core.executablecontent" minOc-curs="0" maxOccurs="unbounded"/>`
`</xsd:sequence>`
`</xsd:group>`
`<xsd:group name="scxml.core.executablecontent">`
`<xsd:choice>`
`<xsd:group ref="scxml.extra.content" minOccurs="1" maxOccurs="1"/>`
`<xsd:element ref="send"/>`
`</xsd:choice>`
`</xsd:group>`
	e = (P, ('STATE'), (P, 'STATE'), 'WAIT', ('duration',Z))

An example of mapping a TASCXML command to an edge in graph G that uses the label PARRALLEL_STATEMENT is shown in Figure 10.

3.3. Step 3: Provide Test-Relevant Services

The outcomes of step 1 and step 2 are test cases in graph G and the capability to translate each test case from a test script language into graph G. The structure of graph G is compatible with existing graph databases such as Neo4J, which guarantees that each graph G can be uploaded to a graph database.

Querying test knowledge is essential to create the facets and to enable the explorative search. The definition of the queries is possible via the provided query language of the underlying graph database. The envisaged user is a test engineer unfamiliar with query languages. Based on his experience, a test case is a sequence of blocks in which the SUT is stimulated and afterwards checked. Hereby, a stimulus is a signal that he wants to

stimulate on the SUT (Semantics). For the test engineer, a stimulus is defined by a signal and a value. Instead, he would see a node in graph G defined by a label and properties (Syntax), which is unfamiliar for a test engineer and impractical. An appropriate query language needs to be built on the definition of statements and blocks instead of nodes and edges. The challenge is that query languages for graphs use exact nodes and edges to define queries. Thus, an abstraction of the natively provided query language is needed to close the gap between the envisaged application scenario and the underlying technology stack.

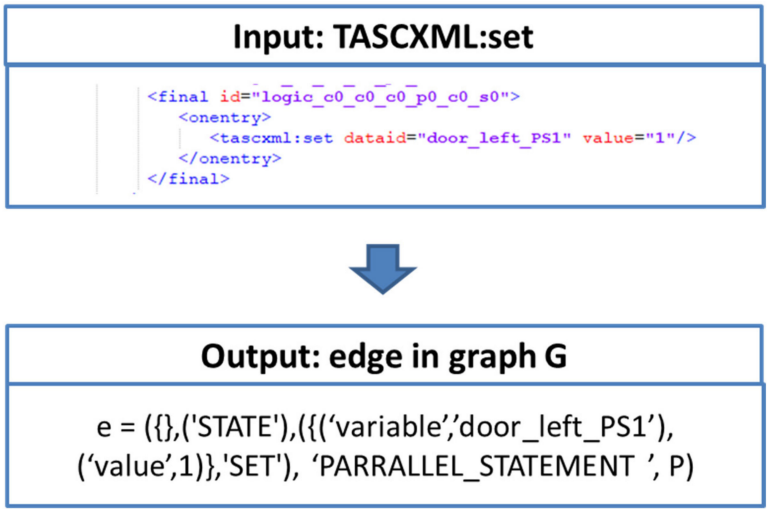

Figure 10. Example for mapping the TASCXML: set to a graph edge.

3.3.1. Pattern Language as the Baseline for Explorative Search

The approach of this paper is to define a query language as a pattern language in which the test engineer defines his/her queries the same way he would create test cases. A query language has been developed for that purpose, and the grammar's basic structure is shown in the following:

grammar TestPatternReduced; (15)

compilationunit: startExpression statements? EOF; (16)

startExpression: 'Pattern' ENDCharacter; (17)

statements: (statement | block)+; (18)

statement: blockingStatement | expectExpression | waitExpression | waitExpressionUntil | lambdaExpression; (19)

blockingStatement: rampExpression | setExpression | verdictExpression; (20)

block: LBRACE blockingStatement + RBRACE; (21)

setExpression: SET qualifiedNameWithOptionalDot (value unit?) ENDCharacter; (22)

expectExpression: EXPECT logicalexpression comparison ENDCharacter; (23)

waitExpression: WAIT value unit? ENDCharacter; (24)

lambdaExpression: MULTIPLICATION qualifiedNameWithOptionalDot (value unit?)? ENDCharacter; ENDCharacter: ';' (25)

SET: 'set'; (26)

EXPECT: 'expect'; (27)

WAIT: 'wait'; (28)

The proposed pattern language defines a corresponding statement for each TASCXML command (represented as V) of graph G. The labels and properties of the graph node **V** are

mapped to a pattern statement. Following this proceeding, specific value assignments for a signal via a *set* or a *ramp* can be easily implemented. An example is shown in Figure 11.

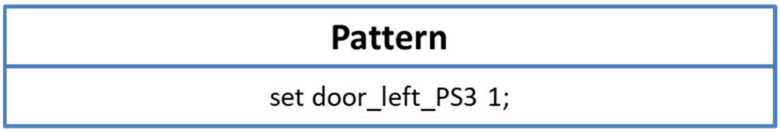

Figure 11. Query for a stimuli I.

It shows that the signal door_left_PS3 sensor is set to 1. In most common test cases, the stimuli are not defined by a single statement but rather by a sequence of statements.

To support this feature, the user can also define a sequence of statements by adding more statements to the pattern definition. In doing so, all statements are modelled as micro-states. It means that the order of the statements is unimportant, and the assumption is that all signals have the assigned value within SUT at the beginning of the following state.

An example is shown in Figure 12. The example also demonstrates a further important feature of the explorative search. The user can add wildcards as a replacement for each token of a statement. It means the test engineer can set the function, the signal, and the value to a wildcard. Within the example, the user specifies all details for the left door and adds the wildcards for the values for the positioning sensors 6 and 7 and the concrete pausing of the test case.

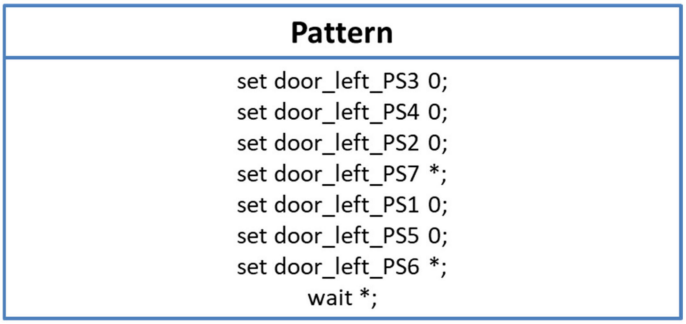

Figure 12. Query for a stimuli II.

3.3.2. Mapping of a Pattern to a Graph Query

The developed pattern language (see Step 3) cannot be executed directly in a graph database such as Neo4J, because the database uses CIPHER as query language. The user defines his pattern in the pattern language, and the developed mapping transforms it into a CYPHER query. In doing so, the mapping method translates each pattern statement into a node whereby the keyword of the statement is mapped to a label, and all parameters also being mapped as node parameters. An example of the mapping is shown in Figure 13.

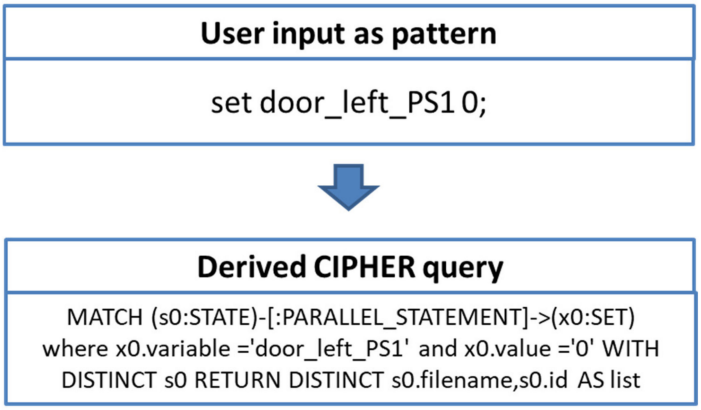

Figure 13. Example for the mapping of a pattern to a CIPHER query.

4. Evaluation of the Test Knowledge Graph

4.1. Application Scenario

The application scenario addresses the test process of an aircraft. More precisely, it focuses on the cabin doors specified in the ATA chapter 52 [30]. The development of a cabin is a special case of the overall development of an aircraft because the cabin is adaptable for each airline. The customisation of the door results in a new cabin configuration that needs to be tested as part of the certification process. For that purpose, test processes are carried out in which the test cases slightly differ between the cabin configurations. Thus, many redundant test activities are performed with the same cabin systems in focus and need a similar system understanding.

The application scenario's test cases follow the typical HIL test structure in which the SUT is first stimulated, the target state is stimulated, and then the SUT reaction is monitored. For this purpose, signals are stimulated and monitored at the SUT. HIL tests are used to test mechatronic systems at the integration and system test level. It does not matter whether it is a car, plane, train, or another complex mechatronic system because the mechatronic system is tested as a black box. Although the selected application scenario deals with the doors, it can be applied 1:1 to other HIL test processes. It means that the test cases in other scenarios would not come from Python (as in the application scenario) but from other test case languages for HIL tests. Accordingly, step 1 would be different, but steps 2 and 3 would be the same.

4.2. Example of Look and Feel of the Pattern Search

The considered test scenario checks whether the doors behave correctly in ordinary and emergency cases. This scenario has been selected because the cabin doors are similarly constructed and integrated into the aircraft's control systems. Thus, test cases for all doors contain patterns and similar proceedings. There is no exact match between the test cases because each door has unique signal names. Before the evaluation is presented, an example of a door's test case, which is fictive and not the real test case, and the results of steps 1–3 are given.

In the following fictive example, the stimulation of a door is represented by fictive seven position sensors, which can take the values 1 (is close by) and 0 (is not close by). This example is a fictive example and simplifies the signals and underlying systems. First, a test case for the stimulation of a cabin door as a test case in CCDL is presented. Subsequently, the outcome of the translation capabilities of steps 1 and 2 as a graph is presented in detail. Finally, example queries for the explorative search are given.

4.2.1. Example of a Test Case as Graph G

The StartUP and the stimulation of such a scenario are shown as a CCDL test case in Figure 14.

```
1  CCD //MDPIGraphV2
2
3  Initial Conditions:
4  {
5  set door_left_PS1 to 0
6  set door_left_PS2 to 0
7  set door_left_PS3 to 0
8  set door_left_PS4 to 0
9  set door_left_PS5 to 0
10 set door_left_PS6 to 1
11 set door_left_PS7 to 1
12 }
13
14 Test Step 1:
15 {
16 set door_left_PS1 to 1
17 wait 5000[ms]
18 }
19
20 Test Step 2:
21 {
22 expect Verdictdoor_left_PS1.value between Verdictdoor_left_PS1.value <=1 and Verdictdoor_left_PS1.value >=1)
23 wait 122001[ms]
24 }
```

Figure 14. Fictive test case of a cabin door.

The test case sets the position sensors 1–5 to the value 0 in the block's initial condition. It means that the door is open. Then, the stimulation of the UUT starts, and the door is closed, but not lot latched and locked. Latched and locked would need to change the value to more than the signal of door_left_PS1 sensor. The test case waits 5 s to ensure the door is closed and the information is shared with all systems. Subsequently, the test case checks whether the door is closed. For that purpose, the statement *expect* in Test Step 2 is used. The representation of this CCDL test case as a graph is shown in Figures 15 and 16.

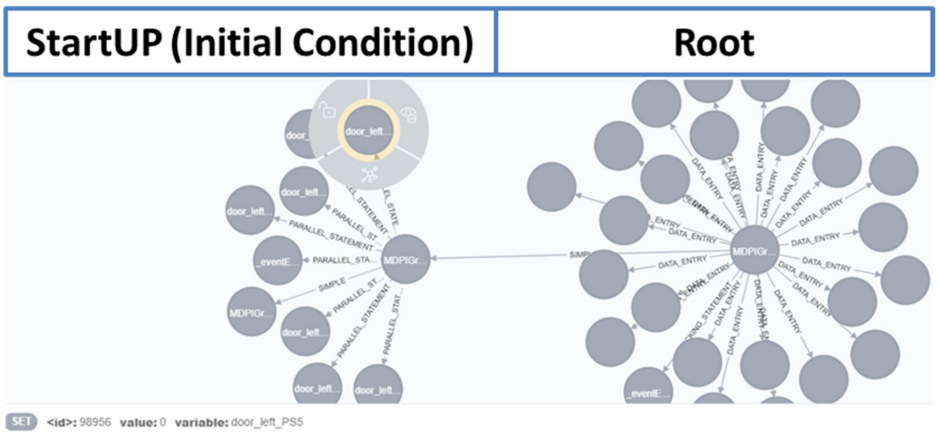

Figure 15. Test case as graph G I.

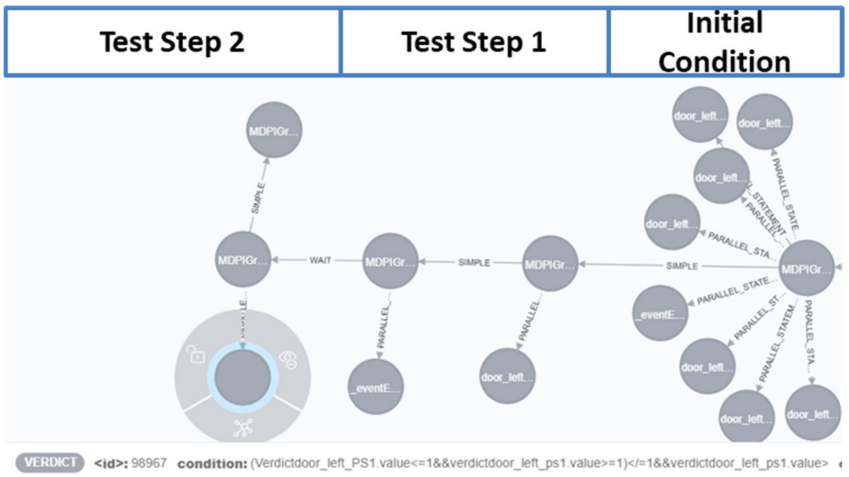

Figure 16. Test case as graph G II.

The graph for the test case starts on the right side with the node MDPIGraph which is the root node of the graph. The data model from the SCXML test case is assigned to the root node as well as the node for the StartUP (Initial Condition in CCDL). The statements of the node for Initial Condition are connected via edges carrying the label "PARALLEL_STATEMENT" as nodes. Each node has the label SET, and the signal name and the value are assigned as properties (**P**). The selected node in Figure 15 defines that the signal door_left_PS5, which is a positioning sensor, is set to 0. The following stimulation, which is contained in Test Step 1 of the CCDL test case, is assigned to another node. For that purpose, the node is connected with the edge defined by the label "SIMPLE", shown in Figure 16. This node has one node connected via the parallel edge and set the signal door_left_PS1 sensor to 1. Subsequently, the wait is added via a wait edge. Finally, an additional node is connected for the implementation of the *expect*. Hereby, the *expect* checks whether the door is open using the door_left_PS1 sensor.

The cabin intercommunication data system (CIDS) controls the cabin systems. It controls the, e.g., the slides/doors, water/waste, smoke indication, air conditioning, etc.

The CIDS controls the doors and shares status information with other systems. The status information of a door is implemented by sensors which can be monitored. The information is applied to classify whether the door is latched, locked or closed. All doors implement the same system states but are described slightly differently. The different signal names are the mentioned differences. The cabin doors are integrated into the overall aircraft safety mechanisms. Accordingly, there are rules on how the doors behave in the respective flight phases. That means that test cases for cabin doors also need to stimulate the SUT in the right way that the CIDS permits the execution of door commands such as opening. For that purpose, a real test case would need to extend the stimulation phase to set the correct flight phase and to define the environmental parameters such as flight altitude or airspeed.

4.2.2. Two Examples of the Explorative Search

First Scenario: Search for the complete stimulation

The first theoretical use case is that a test engineer has already created a test case for the left door, and the test case has been uploaded to the graph database Neo4J. Subsequently, another test engineer needs to create a cabin door test case and know that he has to set the left door open in the StartUP of a test case. For that purpose, he needs to know the corresponding signal name and the value. Thus, he searches for the signal in the System Requirement Document (SRD) and finds out that the signal door_left_PS1 is a position sensor to check whether the door is open or closed. He creates a test case, sets the sensor to 0 in StartUP, and tries executing the test case in a HIL environment. The execution shows that the left door keeps closed in the simulation model. To open the door, additional signals need to be stimulated. In this case, he would need to reread the SRD to identify all dependencies.

In this case, the test engineer could also use the explorative search. For that purpose, he first defines a pattern containing only the stimulation of the door_left_POS1 signal. In doing so, he selects in the user interface (UI) the SET statement and assign the value 0. Subsequently, he/she adds the statement to the current pattern. The result that the Test Advisor performs is shown in Figure 17.

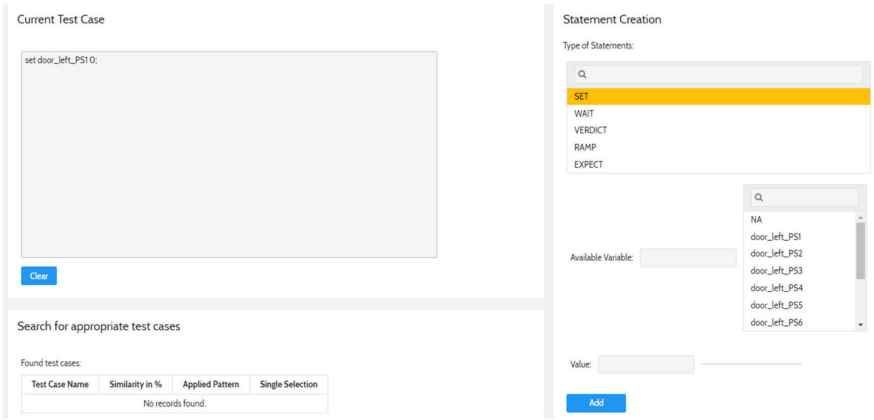

Figure 17. Screenshot of http://testadvisor.eu (accessed on 5 January 2023) how to add a statement to the pattern.

Subsequently, the test engineer would like to query this pattern. For that purpose, the prototype translates the pattern into a CIPHER query. The corresponding query looks for a node connected to the statement's second node. The corresponding query is:

```
MATCH (s0:STATE)-[:PARALLEL_STATEMENT]->(x0:SET)
where x0.variable = 'door_left_PS1' and x0.value = '0' WITH DISTINCT s0
RETURN DISTINCT s0.filename,s0.id AS list
```
(29)

The query results are a list of test cases containing the statement. Based on the selected test case, the test engineer can define how many statements shall be autocompleted. In the example, he decides on an autocompletion until the next wait statement and starts the search. This is because stimulations are separated by wait statements to enable the SUT to set the signals. The start node and the test case are known. The corresponding query is:

$$\text{Match p = (s:STATE)-[*1..10]->(b)-[:WAIT]-> (c) where s.filename = 'MDPIGraphV2' And s.id = 'startup\_c0' RETURN p} \quad (30)$$

The result is the extended pattern which is shown as the screenshot in Figure 18.

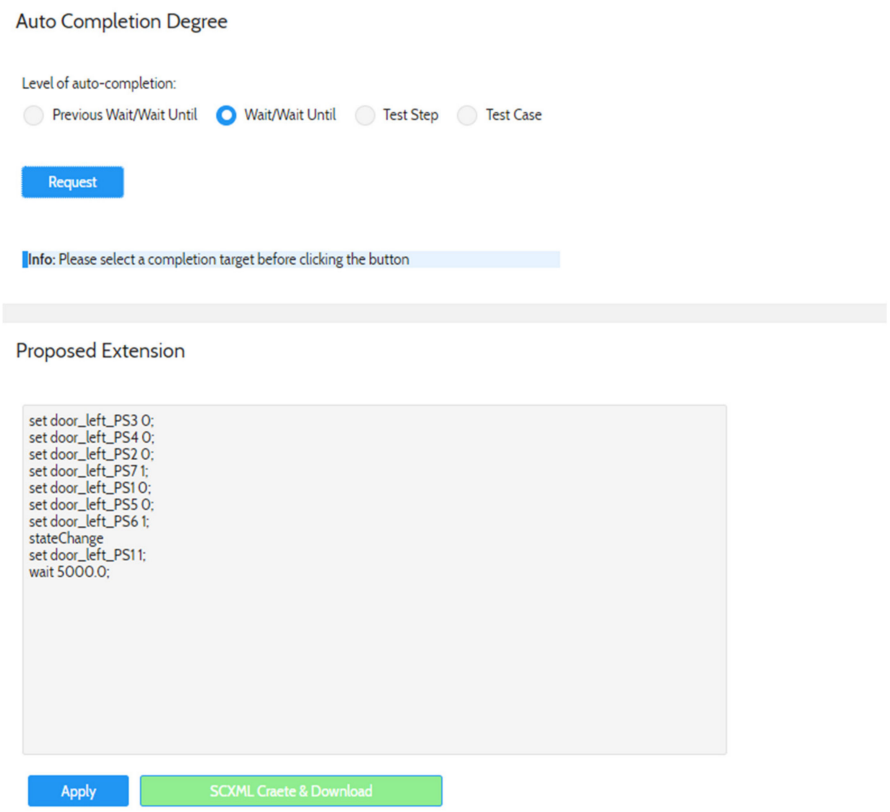

Figure 18. Screenshot of http://testadvisor.eu (accessed on 5 January 2023) for the auto-completion of the pattern with the scope wait/wait until.

Second Scenario: Search for the stimulation based on a requirement

The second theoretical use case is that a test engineer shall check that the door is closeable related to a specific requirement. The signal name Verdictdoor_left_PS1 defines the signal to check whether the door is closed. Based on this, the test engineer can define the verdict statement (In CCDL, it was an *expect* statement) for the test case. The verdict needs the correct stimulation beforehand as a part of a complete test case; thus, the explorative search can search and autocomplete the test case. The scope of the auto-completion would change from Wait/Wait until to the Previous wait/Wait until. All the steps would be similar apart from the CIPHER query for the auto-completion. In this example, the CIPHER query would be:

$$\text{Match p = (s:STATE)<-[:SIMPLE *0..10]-(d)<-[:WAIT]-(b)<-[:SIMPLE]- (c) where s.filename = 'MDPIGraphV2' And s.id = 'logic\_c0\_c1\_c0' RETURN p} \quad (31)$$

4.3. Evaluation

4.3.1. Verification of the Search Capabilities by Similarity Search

The evaluation focuses on steps 2 and 3 of the developed method, because the first step has already been implemented and evaluated successfully [15]. The developed web application, namely Test Advisor (http://testadvisor.eu, accessed on 5 January 2023), implemented the upload of test cases in the graph database Neo4J and the provision of search capabilities for test knowledge. Moreover it is available online as Supplementary Materials.

The prototype was tested in an avionics use case together with Airbus based on real test cases in the scope of the research project AGILE-VT [29]. The test set contains test cases in Python and the transformed test cases in Generic SCXML that are confidential and cannot be published here. In addition, the defined pattern for the evaluation is also confidential because it describes the internal design of an aircraft door. To enable the replication of the pattern search of the above shown example, a fictive and abstracted version of the test case for the left door is published within the web application http://testadvisor.eu. Interested readers are welcome to request a user account from the first author and evaluate the search capabilities.

To keep every step in the evaluation, despite the confidentiality of the test cases, traceable and comprehensible, the complexity of the use case was chosen appropriately. The evaluation case addresses the opening and closing of an aircraft door on the ground. A fictive and abstracted version of the beginning of the left door test case is shown in Figure 14. The prerequisites of the test set, the execution of the evaluation and the evaluation results are preliminary work and have been published in [31]. The evaluation is still valid according to the representation of a test case as a graph (for the stimulation) and the pattern search. Since then, the methods for steps 1, 2 and 3 have been adapted to the current Generic SCXML version and the support of the TASCXML statements. In the following, the results are shown whereby more details related to the test set and the results are given compared to [31]:

- All test cases are provided in Generic SCXML, which is a derivate of SCXML for test cases. For that purpose, the original test cases have been translated from Python to Generic SCXML using the step 1 solution;
- The test corpus includes test cases for different doors with the same stimuli and checking the same behaviour. They only differ in specific signal names;
- The test cases for *ordinary* and *emergency* operations are similar apart from the emergency-specific signal values.

The test bed contains the following test cases, which are shown in Table 5.

Based on the prerequisites, a snippet has been defined for the first door, which contains TASCXML: set statements for the position sensors and is analogical to the first stimulus contained in the StartUP state of the real Generic SCXML test case. It includes five contains TASCXML: set and a wait statement. The snippet's content describes the initial state of an ordinary door test case. The intention of the snippet is comparable to the example of the initial condition in Figure 14. The real applied test cases use different signal names, implement more logic and include more TASCXML:set statements for the stimulation of the SUT, but the principle of the stimulation is comparable. In addition, no wildcards have been used in the snippet. Subsequently, the snippet has been searched in the knowledge graph stored in Neo4J. The outcome of a search is a set of results. All the results have the same similarity for each result, the name of the test case and locations inside the test case is given.

Table 5. Applied test cases as input for the knowledge graph.

Test Case	Length in Python Lines	Length in SCXML Lines	Length as Complete Pattern Lines
Left Door 1, ordinary	58 lines but importing 7 libraries (all libraries have 15881 lines)	18,434	Unknown
Left Door 1, emergency	59 lines but importing 7 libraries (all libraries have 15881 lines)	20,520	1001
Right Door 1, ordinary	58 lines but importing 7 libraries (all libraries have 15881 lines)	18,434	Unknown
Right Door 1, emergency	59 lines but importing 7 libraries (all libraries have 15881 lines)	20,520	Unknown
Left Door 2, ordinary	58 lines but importing 7 libraries (all libraries have 15881 lines)	18,434	Unknown
Left Door 2, emergency	59 lines but importing 7 libraries (all libraries have 15881 lines)	20,520	Unknown
Right Door 2, ordinary	58 lines but importing 7 libraries (all libraries have 15881 lines)	18,434	Unknown
Right Door 2, emergency	59 lines but importing 7 libraries (all libraries have 15881 lines)	20,520	Unknown
Left Door 3, ordinary	58 lines but importing 7 libraries (all libraries have 15881 lines)	18,434	Unknown
Left Door 3, emergency	59 lines but importing 7 libraries (all libraries have 15881 lines)	20,520	Unknown
Right Door 3, ordinary	58 lines but importing 7 libraries (all libraries have 15881 lines)	18,434	Unknown
Right Door 3, emergency	59 lines but importing 7 libraries (all libraries have 15881 lines)	20,520	Unknown
Left Door 4, ordinary	58 lines but importing 7 libraries (all libraries have 15881 lines)	18,434	Unknown
Left Door 4, emergency	59 lines but importing 7 libraries (all libraries have 15881 lines)	20,520	Unknown
Right Door 4, ordinary	58 lines but importing 7 libraries (all libraries have 15881 lines)	18,434	Unknown
Right Door 4, emergency	59 lines but importing 7 libraries (all libraries have 15881 lines)	20,520	Unknown

The similarity is defined based on the match between the test case and the original pattern. If the pattern is an exact match, no wildcards need to be added, and the similarity is 100%. If there is no match to the original pattern, the pattern is manipulated, and one wildcard is added. The wildcard replaces the value of a statement. If there is still no match, another value is replaced by a wildcard. If all values have been replaced with wildcards iteratively and there is still no match, then the variables are replaced iteratively by wildcards. If there is still no match, the type of the statement is replaced iteratively by wildcards. Finally, there is a match if wildcards have replaced all tokens of the statements. The number of added wildcards determines the similarity of the found pattern. For that purpose, the (tokens' number of statements -wildcards) are divided by the tokens' number of statements to obtain a percentage value. The replacement order in which a wildcard shall replace parts of a pattern can be configured.

The recursive adaption of pattern results exactly in a search result in which all found test cases have the same similarity. No further criteria are defined to specify the relevance or enable the prioritisation of the test cases within the search results. Moreover, the evaluation does not cover whether the found test cases, which are correct according to the pattern/Cypher query, make sense for the user. The evaluation of whether the user expectation is met (such as mean average precision (mAP)) is not part of the article.

The outcome of the above-mentioned evaluation scenario was that the snippet (comparable to the Initial condition of Figure 14) was found in the first test step in all door-relevant test cases:

- The match was 100% for the first door test case. The result returned the exact test case name and the state id of the state in which the pattern was contained;
- The similarity was lower than 100% for the other test cases. The lower similarity value resulted from the different signal names for the other doors. If the user selects a replacement order as described above, the similarity is 33,3% for test cases of the same operational mode. In addition, if the user selects the replacement order to first substitute the signal names with wildcards, the similarity is 66,6% having the same

operational mode. The result returned a list of the other 14 test cases defining the test case name and the state id where the pattern starts;

- The match between both variants of the first test case (emergency, ordinary operation) was also lower than 100% because both test cases differ in the signals for the emergency operation. To evaluate this capability, the test case of the first left door for the operational modus ordinary has been translated to a pattern and has been searched inside the Neo4J database. The pattern had a length of 1001 statements, and the calculated similarity to the emergency test case of the first left door was 96.81%;
- The right region of the pattern was found in every test case.

The evaluation demonstrated that test cases are discoverable based on a snippet in a knowledge graph, whereby the validated length of the snippet can range up to 1001 statements, including wildcards. Similar results are obtained if the user defines his pattern based on set, verdicts or waits. Thus, the content of a test case is accessible and explorable by its meaning instead of by searching for keywords in the test case syntax.

The quality of the presented pattern search could not be compared with other solutions. The reason is that the authors are unaware of any solutions that can aggregate test cases from different test script languages in the scope of HIL tests, apply them as one data source and search within this data source. Thus, the authors are not aware of any solution that allows a pattern search beyond the limits of a test script language.

Concerning the pattern search within the boundaries of a test script language, some solutions can search the syntax using regular expressions or NLP approaches. These solutions are not applicable for two reasons:

1. The original test case is implemented in Python and uses Python libraries to enfold its capabilities. Any string-matching method can only work on the syntax level and could find, for example, the set statement in three different files: At least one file for the implementation, one file of concrete usage and one for the specific configuration of the signal. All three locations define one statement of a pattern. The question is also what should be returned as a search result. Subsequently, if the pattern is extended to two statements, the string-matching method is not applicable anymore. To enable any search for more than 1 statement, the test case and the libraries have to be merged into one single file already contained in step 1 of the solution;
2. The definition of a pattern on the syntax level must address a lot of variants because the syntax differs between the test cases while having the same semantics. Moreover, one particular detail of test cases increases the complexity of the supported variants. It should be noted that the executed TASCXML commands are described within Python or within an SCXML state or test script in a sequence, but the semantics here are a set. This means that the order of the TASCXML commands is irrelevant, and this fact should also be considered for regular expressions. This means two different sequences of statements are semantically similar if they include the same set of TASCXML commands. To do this, the individual variations would have to be linked with the or operator, which makes the regular expressions very long and not efficient. Moreover, the built-in search functions in development environments focus on the programming language's syntax, not semantics, and are, therefore, not applicable. In summary, no solutions are known which enable the search over semantics in test cases from multiple test script languages.

4.3.2. Validation of the Search Capabilities by a Questionnaire

The goal of the validation is to ensure that the developed pattern search meets the operational needs of test engineers. For that purpose, this type of evaluation needs to be performed by a questionnaire. The questionnaire focuses on the presented search scenarios, namely {First Scenario: Search for the complete stimulation and Second Scenario: Search for the stimulation based on a requirement}.

For each scenario, the user needs to use the online version of the demonstrator (https://testadvisor.eu, accessed on 5 January 2023) to try out the search. After a user has

executed a scenario, the user must fill out the related questionnaire's section. With this, the questionnaire explores whether the search scenario is applicable to the developed pattern search and how useful the search results are. For that purpose, the user must answer four questions, namely {Could you find the complete stimulus related to the stimulation of <search query>?, Could you use the search result to implement your own stimulus in a test case in your test case language?, How complete is the information for the creation of a similar test case?, Could you find similar results with your existing search methods?}.

Finally, the user needs to answer whether he is familiar with the creation of test cases for HIL test processes. The questionnaire is online (https://forms.gle/kJkiAFL2mJUVeaCt5, accessed on 5 January 2023) available, and all interested users are welcome to try out the application scenarios and fill out the questionnaire. In the scope of the linked research projects and corresponding target audience, this type of evaluation has not been performed, and it is no longer feasible. This means that there are no validation results available. To reach a validation, the goal is to perform an open validation based on the two application scenarios using the example data and an ongoing evaluation by the questionnaire. The questionnaire is shown as screenshots in Figures 19–21.

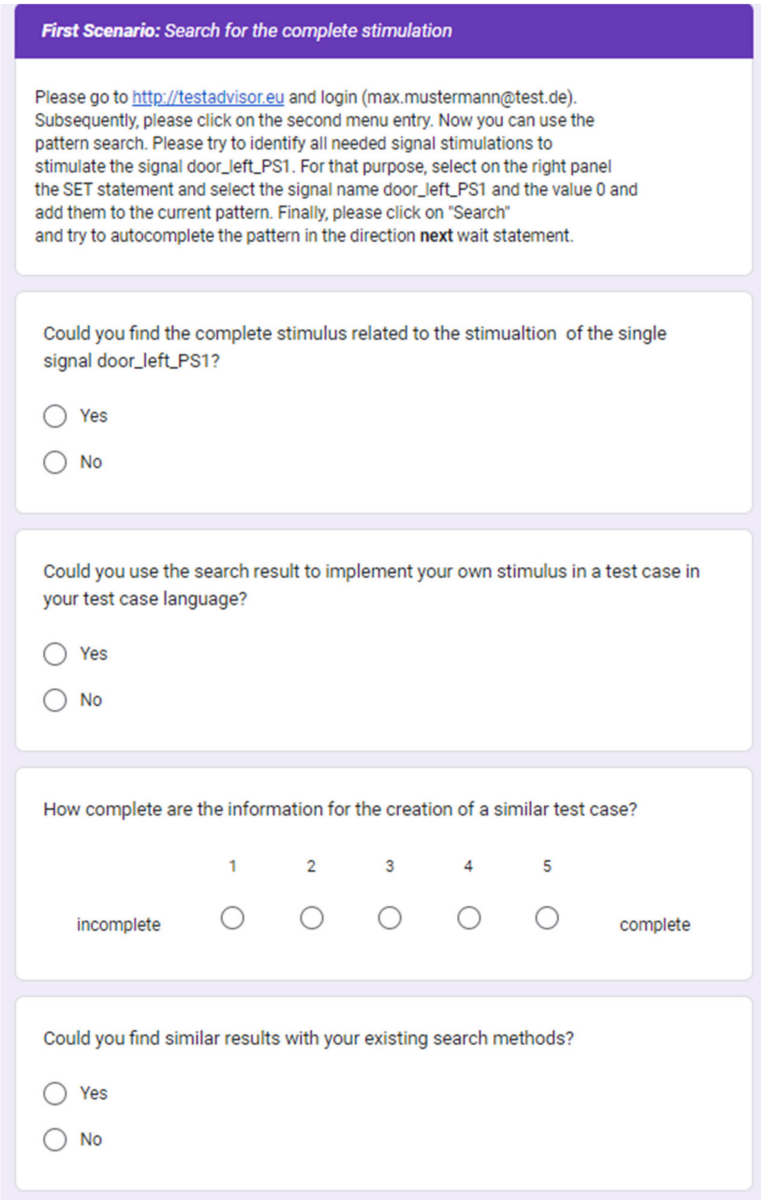

Figure 19. Page one from the questionnaire.

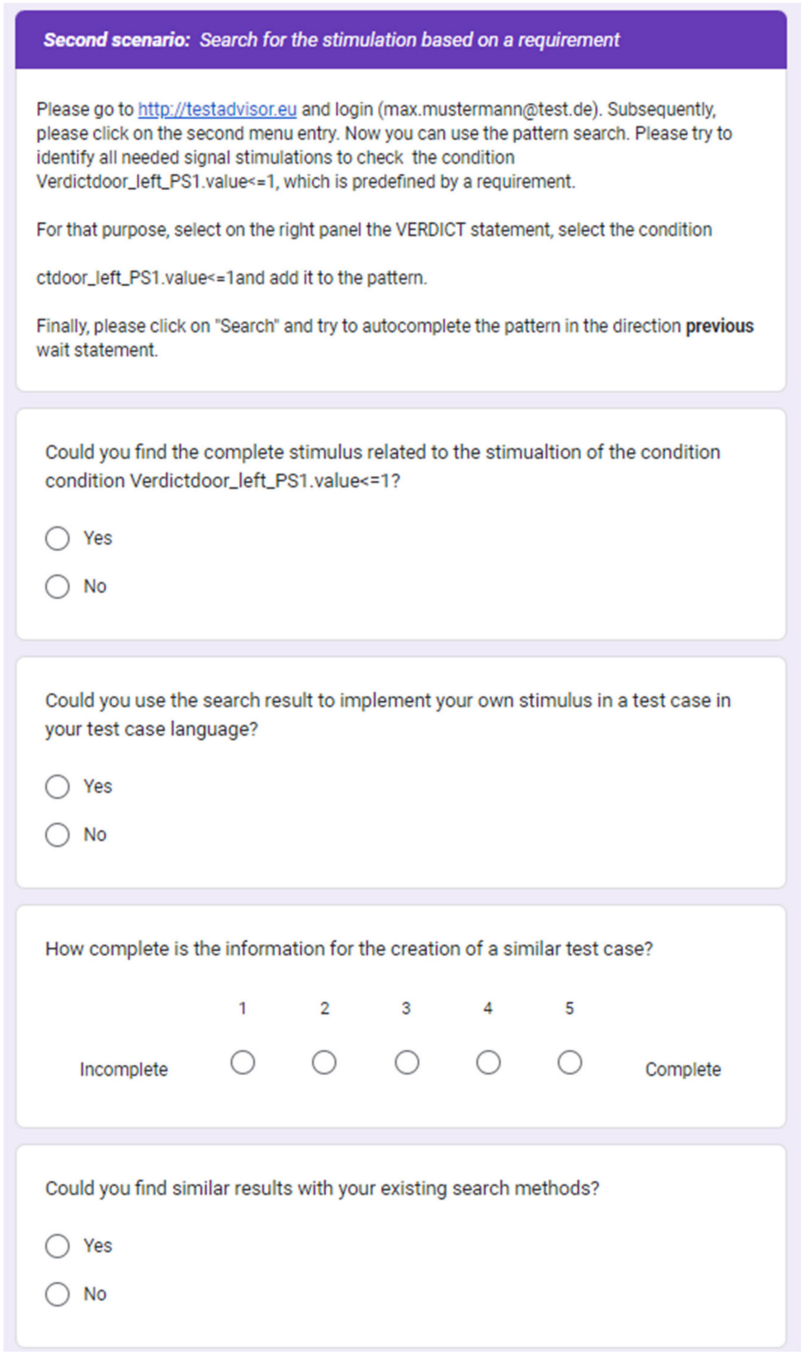

Figure 20. Page two from the questionnaire.

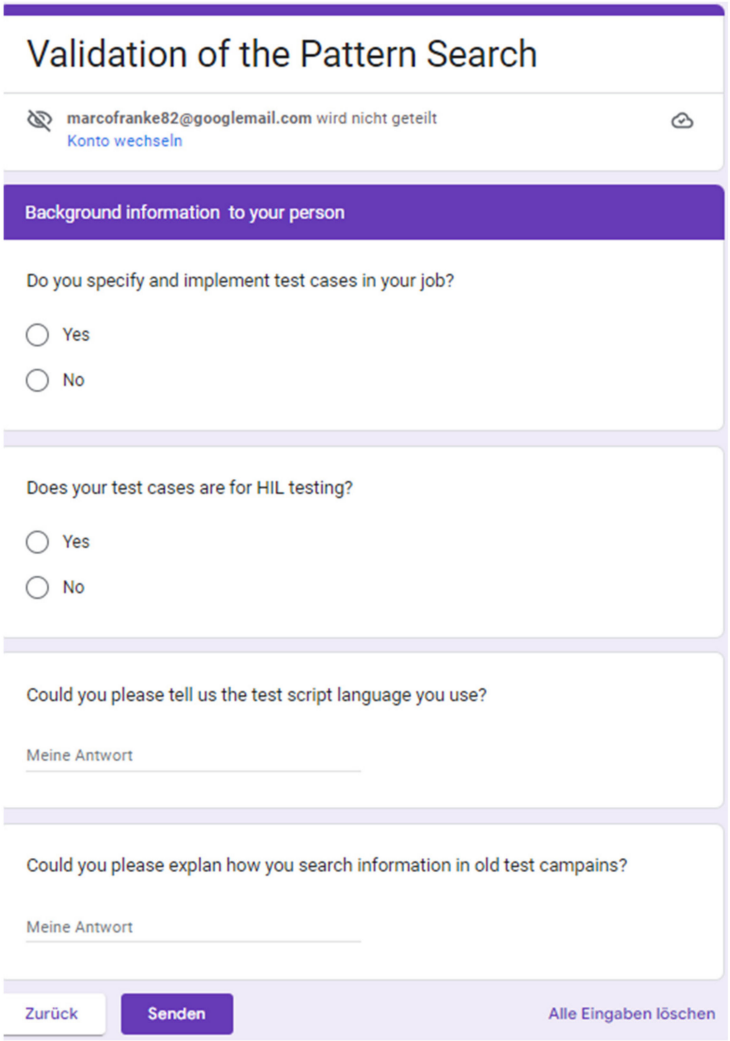

Figure 21. Page three from the questionnaire.

5. Discussions

5.1. Challenges and Findings

The test processes are heterogeneous in a supply chain. The heterogeneity is given because the test processes within a supply chain test different systems on different test levels and need to fulfil different system element requirements. The heterogeneity becomes a challenge as soon as test knowledge shall be exchanged over the boundaries of test processes, considering the different test script languages.

The need for the aggregation of test cases into test knowledge is there. The developed method enables the extraction of test knowledge from the test case's syntax of different test script languages (data view) into a knowledge graph (as information view). The semantic uplifting contains information of stimuli, verdicts, and control of timing. This information content covers the functional parts of a test case. For that purpose, the knowledge graph's model G for test knowledge was specified, and the required transformation capabilities were implemented. Thus, aggregating test knowledge populated from test cases is feasible and achieves the ATHENA interoperability level data/information. A query language was developed so that the knowledge graph does not become another data sink in product lifecycle management. This query language was optimised for the facetted and explorative search. In addition, the query language was tailored to the test engineer so that he could use it directly with his skills.

The evaluation demonstrated that test cases provided in SCXML could be transformed and imported to the graph database Neo4J, whereby the functional aspects of a test case

are preserved. The applicability of SCXML as an interoperability model for test cases was confirmed and developed by Rasche et al. [7]. In addition, the evaluation demonstrated that the test knowledge could be defined and searched over all uploaded cabin door-related test cases. With this, all test processes that focus on HIL-based test processes are applicable.

5.2. Alignment with Existing Studies

The application scenario of a knowledge graph is by Hogan [19] to "define a knowledge graph as a graph of data intended to accumulate and convey knowledge of the real world, whose nodes represent entities of interest and whose edges represent potentially different relations between these entities". The paper addresses the test knowledge contained in test cases as the real-world knowledge that shall be formalised. The real entities are the signals of a complex mechatronic system that can be stimulated or checked. Moreover, the edges define the relations between the signals related to their timing. With this, the sequential and parallel timing of signals can be modelled. Thus, the application domain of testing fits Hogan's expected application scenario and definition.

The required graph model for the formulation of the test knowledge uses the labels of a directed edge-labelled graphs [19] and the properties of a property graphs [19]. These graph types are supported by common graph databases such as Neo4J, and ensure the transferability of the research results for ongoing research and commercialisation.

The selected interoperability approach for test cases as part of step 1 achieves the interoperability on the ATHENA interoperability level data/information. For that purpose, SCXML [6,7] has been selected as an interoperability model for test cases. With this, test cases in SCXML are the input for the import of test cases into the knowledge graph. The continuous development of SCXML as an interoperability model will increase the amount of supported test script languages. Thus, the applicability of the facetted and explorative search will continue to grow in the heterogeneous test processes, and benefits from a growing amount of translation approaches for translating test cases into SCXML. The presented method is still applicable if SCXML cannot be established as the standard interoperability model of test cases. SCXML uses state charts as the underlying technology. Most of the test models of the application domain of model-based testing use state charts or comparable models to represent test cases [8,10]. The transformation of these test models into SCXML is possible because there are available generic models to model transformation solutions, such as OPM modelling tool [32] or transformation languages, such as query view transformation (MOF QVT).

5.3. Implications for Practice

The article's aim was to aggregate test knowledge into a knowledge graph. Initially, data sources are filled with test cases from different test script languages and spread over the supply chain test processes. These data sources already contain countless test cases from different aircraft types ranging from A220 to A380 and corresponding variants. While the used test case script languages have changed over time, the basics of the flight dynamics and the control systems of an aircraft have not changed. This means that test knowledge for joint usual flight manoeuvres, including the right stimulus and the expected behaviour, already exists, and the developed search method can make it searchable and applicable. In doing so, despite the heterogeneity of test script languages, the developed method enables the aggregation of test cases and feeds test knowledge for information-driven services. The implemented information-driven service provides a search offering facetted and explorative search modus. A graph model, the integration into Neo4J and a test engineer-friendly query language were developed for that purpose.

Concerning the application of the proposed solution in practice, the solution is applicable within the supply chain's test processes independently of the used test script language. The benefits are that test knowledge can be added to the holistic view and can be used directly in the design, specification, and implementation of test cases. The added value for the daily business of a test engineer is presented as an example in the following.

After several test campaigns for his product family, an experienced test engineer acquires the test knowledge that is already part of the knowledge graph G. In this way, he can, for example, recap all dependent signals for a given signal that gains a valid stimulation. Until now, young test engineers used experienced test engineers as a source of knowledge. The developed approach reduces this dependency. Young colleagues can now access this knowledge from the knowledge base instead of the experienced test engineer and use it directly for their test campaigns by making explorative search queries. The described use case is just an example, and a couple of more use cases according to the optimisation within HIL test processes are possible.

5.4. Limitations and Future Research

The developed method requires step 1 as the precondition for the aggregation (step 2) and searches capabilities (step 3). As long as the test cases from a given domain can be translated to SCXML, they can also be represented as a graph. The limitation results from the coverage of the translatable functional aspects of the test case, according to SCXML. If the definition of the TASCXML commands changes, graph G, including the transformation functions **f** and **f2**, must also be adjusted. Test cases written in Excel sheets or natural language are manual tests and cannot be imported into the presented graph G using steps 1 and 2. For that purpose, a preliminary step 0 using natural language processing (NLP) is necessary to translate the test cases into a test script language. Apart from the technical limitation, the intellectual property right (IPR) and right of use must be considered. The OEM and all suppliers develop test cases in different test script languages, and only the aggregation of all test cases as a knowledge graph would achieve the product's complete picture. The challenge is that the IPR of each uploaded test case, independent of its representation form, needs to be considered. This means that the knowledge graph must consider a set of IPRs and needs to implement an appropriate rights and roles concept. These circumstances largely determine the extent to which the developed method can be used in the operative business of a supply chain.

Future work shall focus on a more detailed evaluation of whether the pattern search works correctly from the user's perspective. Even though the derived CIPHER queries always find all test cases, the user's point of view is missing. With this, a test case could be found by the CIPHER query, but the test case makes no sense to the user's needs. To evaluate potential discrepancies, metrics such as mean average precision (mAP) can be applied. For that purpose, a public test scenario in combination with a public questionnaire could identify the needed recall and precision values. For that purpose, the test scenario and the questionnaire are online, and test users are needed.

A second direction of future work is related to implicit patterns. A populated graph database does not only support the search for explicit patterns. Implicit patterns, i.e., patterns the test engineer is unaware of, are valuable. Every implicit pattern found can be used to build up test cases modularly, making them more sustainable. Future work should focus on the modularisation of test cases. The aim is to research which sequences of statements are repeated in the graph. Such repetitive sequences are suitable for modularising a test case by introducing macro functions. For this, the search would have to be supplemented with a new mode that can search for isomorphic subgraphs.

A populated graph database describes the aircraft behaviour entirely as a system under test. This test knowledge can also be applied to train supervised learning algorithms. Potential use cases are the calibration of simulation models against operational data from the airline fleet or the training of surrogate models as replacements for simulation models. The proper selection of stimuli and expected behaviour for the training and verification data is essential for both use cases. The knowledge graph can be used to generate the input data for that purpose.

6. Conclusions

The presented approach enables the semantic uplifting of test cases from several different test script languages into a single knowledge graph. By doing so, we created a joint understanding and a holistic view of the mechatronic system as a system under test. So far, this test knowledge has existed only in the minds of experienced test engineers with no digital knowledge graph and no automated search capabilities. We developed a query language that enables a search for stimulation, verdicts, and timing within many test cases. The representation of test cases as knowledge graphs and the intended querying capabilities were successfully tested on cabin door test cases of an aircraft.

Supplementary Materials: The search and auto completion tool is available at: http://testadvisor.eu/.

Author Contributions: Conceptualisation, M.F. and K.-D.T.; writing—original draft preparation, M.F.; writing—review and editing, B.E.; writing—review and editing, K.-D.T.; supervision, K.-D.T. All authors have read and agreed to the published version of the manuscript.

Funding: This research has been funded by Federal Ministry for Economic Affairs and Climate Action (BMWK) in the projects STEVE (project number 20Y1301G) and AGILE-VT (project number 20X1730D).

Data Availability Statement: Not applicable.

Acknowledgments: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Bolton, W. *Mechatronics: Electronic Control Systems in Mechanical Engineering*; Longman Scientific & Technical: Harlow, UK, 1995; ISBN 0582256348.
2. Histand, M.B.; Alciatore, D.G. *Introduction to Mechatronics and Measurement Systems*, international ed.; WCB/McGraw-Hill: Boston, MA, USA, 1999; ISBN 0071163778.
3. Campetelli, A.; Broy, M. Modelling Logical Architecture of Mechatronic Systems and Its Quality Control. In *Automotive Systems Engineering II*; Winner, H., Prokop, G., Maurer, M., Eds.; Springer International Publishing: Cham, Switzerland, 2018; pp. 73–91. ISBN 978-3-319-61605-6.
4. Braid, D.M.; Johnson, C.W.; Schillinger, G.D. An integrated test approach for avionics system development. In Proceedings of the 2001 IEEE/AIAA 20th Digital Avionics Systems Conference, Daytona Beach, FL, USA, 14–18 October 2001; IEEE: Piscataway, NJ, USA; pp. 9B2/1–9B2/9, ISBN 0-7803-7034-1.
5. Mitra, R.; Jayakumar, S.S.; Kishore, K.R. Proof of Concept on Application of Formal Methods at different phases of development cycle of Avionics Software A Landscaping. *INCOSE Int. Symp.* **2016**, *26*, 329–345. [CrossRef]
6. Franke, M.; Meyer, V.; Rasche, R.; Himmler, A.; Thoben, K.-D. Interoperability of Test Procedures Between Enterprises. In *Enterprise Interoperability VIII*; Popplewell, K., Thoben, K.-D., Knothe, T., Eds.; Smart Services and Business Impact of Enterprise Interoperability: Cham, Switzerland; Springer International Publishing: New York, NY, USA, 2019; pp. 177–188. ISBN 978-3-030-13693-2.
7. Rasche, R.; Himmler, A.; Franke, M.; Meyer, V.; Klaus-Dieter, T. (Eds.) Interfacing & Interchanging–Reusing Real-Time Tests for Safety-Critical Systems. In Proceedings of the 2018 AIAA Modeling and Simulation Technologies Conference, Kissimmee, FL, USA, 8–12 January 2018; p. 0123.
8. Franke, M.; Gerke, D.; Hans, C.; Thoben, K. Functional System Verification. In *Air Transport and Operations: Proceedings of the Third International Air Transport and Operations Symposium 2012*; IOS Press: Amsterdam, The Netherlands, 2012; p. 36.
9. Franke, M.; Krause, S.; Thoben, K.-D.; Himmler, A.; Hribernik, K.A. Adaptive Test Feedback Loop: A Modeling Approach for Checking Side Effects during Test Execution in Advised Explorative Testing. In *SAE Technical Paper Series*; AeroTech, MAR. 17, 2020; SAE International: Warrendale, PA, USA, 2020.
10. Peleska, J. Industrial-Strength Model-Based Testing-State of the Art and Current Challenges. *Electron. Proc. Theor. Comput. Sci.* **2013**, *111*, 3–28. [CrossRef]
11. Singhal, A. Introducing the Knowledge Graph: Things, Not Strings. Google [Online]. 16 May 2012. Available online: https://blog.google/products/search/introducing-knowledge-graph-things-not/ (accessed on 9 January 2023).
12. Guarino, N. *Formal Ontology in Information Systems: Proceedings of the First International Conference (FOIS'98), 6–8 June, Trento, Italy*; IOS Press: Amsterdam, The Netherlands, 1998; ISBN 9789051993998.
13. Bizer, C.; Lehmann, J.; Kobilarov, G.; Auer, S.; Becker, C.; Cyganiak, R.; Hellmann, S. DBpedia-A crystallisation point for the Web of Data. *J. Web Semant.* **2009**, *7*, 154–165. [CrossRef]

14. *IEEE Std 610.12-1990*; IEEE Standard Glossary of Software Engineering Terminology. IEEE Std 610.12-1990; IEEE: Piscataway, NJ, USA, 1990; pp. 1–84.
15. Franke, M.; Thoben, K.-D. Interoperable Test Cases to Mediate between Supply Chain's Test Processes. *Information* **2022**, *13*, 498. [CrossRef]
16. Bonatti, P.A.; Decker, S.; Polleres, A.; Presutti, V. Knowledge graphs: New directions for knowledge representation on the semantic web (dagstuhl seminar 18371). In *Dagstuhl Reports*; Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik: Wadern, Germany, 2019.
17. Ehrlinger, L.; Wöß, W. Towards a definition of knowledge graphs. *SEMANTiCS* **2016**, *48*, 2.
18. Bergman, M.K. A Common Sense View of Knowledge Graphs. Available online: https://www.mkbergman.com/2244/a-common-sense-view-of-knowledge-graphs/ (accessed on 31 August 2022).
19. Hogan, A.; Blomqvist, E.; Cochez, M.; d'Amato, C.; Melo, G.D.; Gutierrez, C.; Kirrane, S.; Gayo, J.E.L.; Navigli, R.; Neumaier, S.; et al. Knowledge Graphs. *ACM Comput. Surv.* **2021**, *54*, 1–37. [CrossRef]
20. Uschold, M.; Gruninger, M. Ontologies: Principles, methods and applications. *Knowl. Eng. Rev.* **1996**, *11*, 93–136. [CrossRef]
21. Zhang, Y.; Zhu, H. Ontology for Service Oriented Testing of Web Services. In Proceedings of the 2008 IEEE International Symposium on Service-Oriented System Engineering, Jhongli, Taiwan, 18–19 December 2008; pp. 129–134.
22. Bezerra, D.; Costa, A.; Okada, K. SWTOI (software test ontology integrated) and its application in Linux test. In *International Workshop on Ontology, Conceptualization and Epistemology for Information Systems, Software Engineering and Service Science*; Elsevier: Amsterdam, The Netherlands, 2009; pp. 25–36.
23. Pilorget, L. (Ed.) *Testen von Informationssystemen: Integriertes und Prozessorientiertes Testen*; Vieweg+Teubner Verlag: Wiesbaden, Germany, 2012; ISBN 978-3-8348-1866-9.
24. International Software Testing Qualifications Board. Available online: https://www.istqb.org/ (accessed on 11 November 2022).
25. Pierce, B.C. Types and programming languages. In *Bulletin of Symbolic Logic*; MIT Press: Cambridge, MA, USA, 2002; pp. 213–214.
26. Jifa, G. Data, Information, Knowledge, Wisdom and Meta-Synthesis of Wisdom-Comment on Wisdom Global and Wisdom Cities. *Procedia Comput. Sci.* **2013**, *17*, 713–719. [CrossRef]
27. Bellinger, G.; Castro, D.; Mills, A. Data, Information, Knowledge, and Wisdom. Available online: http://www.systemsthinking.org/dikw/dikw.htm (accessed on 9 January 2023).
28. Berre, A.-J.; Elvesæter, B.; Figay, N.; Guglielmina, C.; Johnsen, S.G.; Karlsen, D.; Knothe, T.; Lippe, S. The ATHENA interoperability framework. In *Enterprise Interoperability II*; Springer: Berlin/Heidelberg, Germany, 2007; pp. 569–580.
29. dSPACE GmbH. *AGILE-VT-dSPACE-Agiles Virtuelles Testen der Nächsten Generation für die Luftfahrtindustrie von dSPACE im Verbund AGILE-VT: Schlussbericht-Öffentlich: BMWi-Verbundprojekt im Rahmen des Luftfahrtforschungsprogramms LuFo V-3: Berichtszeitraum: 1. Oktober 2017-30. Juni 2021*; dSPACE GmbH: Paderborn, Germany, 2021.
30. Federal Aviation Administration. Federal Aviation Administration Joint Aircraft System/Component Code Table and Definitions. Available online: https://air.flyingway.com/books/jasc-codes.pdf (accessed on 8 December 2022).
31. Franke, M.; Hribernik, K.A.; Thoben, K.-D. Interoperable Access to Heterogeneous Test Knowledge. *SSRN J.* **2020**. [CrossRef]
32. Shani, U.; Franke, M.; Hribernik, K.A.; Thoben, K.-D. Ontology mediation to rule them all: Managing the plurality in product service systems. In Proceedings of the 2017 Annual IEEE International Systems Conference (SysCon), Montreal, QC, Canada, 24–27 April 2017; IEEE: New York, NY, USA; pp. 1–7. [CrossRef]