Supporting the Validation of Structured Analysis Specifications in the Engineering of Information Systems by Test Path Exploration

Torsten Bandyszak[1], Mark Rzepka[2], Thorsten Weyer[1] and Klaus Pohl[1]
[1]*Paluno – The Ruhr Institute for Software Technology, University of Duisburg-Essen, Essen, Germany*
[2]*Protel Hotelsoftware GmbH, Dortmund, Germany*

Keywords: Requirements Validation, Reviews, Structured Analysis, Data Flow Diagrams, Test Paths, Test Cases.

Abstract: Requirements validation should be carried out early in the development process to assure that the requirements specification correctly reflects stakeholder's intentions, and to avoid the propagation of defects to subsequent phases. In addition to reviews, early test case creation is a commonly used requirements validation technique. However, manual test case derivation from specifications without formal semantics is costly, and requires experience in testing. This paper focuses on Structured Analysis as a semi-formal technique for specifying information systems requirements, which is part of latest requirements engineering curricula and widely accepted practices in business analysis. However, there is insufficient guidance and tool support for creating test cases without the need for using formal extensions in early development stages. Functional decomposition as a core concept of Structured Analysis, and the resulting distribution of control flow information complicates the identification of dependencies between system inputs and outputs. We propose a technique for automatically identifying test paths in Structured Analysis specifications. These test paths constitute the basis for defining test cases, and support requirements validation by guiding and structuring the review process.

1 INTRODUCTION

Early validation of requirements artifacts is a crucial task in software engineering, since the costs for finding and correcting defects is more expensive in later phases (Boehm and Basili, 2001). Validating requirements aims at discovering requirements quality issues, and assuring that stakeholders and requirements engineers share the same understanding of the system to be developed (Dzida and Freitag, 1998). To this end, specification reviews or inspections are widespread validation techniques usually involving requirements engineers and stakeholders, e.g., domain experts or users (Kotonya and Sommerville, 1998).

Creating test cases based on requirements can be employed to support requirements validation. For instance, perspective-based reviews (Shull et al., 2000) from a tester's point of view guide the reviewers in creating a set of test cases. Test case definition requires a deep understanding of the specification, and thus helps identifying issues such as incorrectness, ambiguities, inconsistencies, or incompleteness (Kotonya and Sommerville, 1998; Denger and Olsson, 2005). However, unless formal specification languages and automatic test case generation techniques are used, it also requires some experience in testing, and might involve costs for training. Tester participation in reviews is advocated (Graham, 2002), but often not feasible, and poses challenges due to different backgrounds (Uusitalo et al., 2008). Thus, it is desirable to facilitate test case generation from early, semi-formal specifications by using tools, so as to reduce the required knowledge and support stakeholders without testing expertise.

In this paper, we focus on Structured Analysis as proposed by DeMarco (1979), and do not consider e.g. formal or real-time extensions. Structured Analysis is taught to professionals as part of widely accepted current requirements engineering curricula, e.g. (IREB, 2012). The key concept of functional decomposition, which has been adopted in many modern approaches, incorporates Data Flow Diagrams (DFDs) to partition a system. Especially DFDs are still widely used today (Pressman, 2010), and constitute a commonly used notation for modeling information systems (Giaglis, 2001). Both DFDs and functional decomposition are recommended as best-practice techniques for business analysis (IIBA, 2009).

However, system partitioning results in a set of processes and associated process specifications that are distributed among several abstraction layers, which complicates the derivation of test cases (Roper and Bin Ab Rahim, 1993; Roper, 1994). Particularly, it is not easy to determine test paths through the entire specification due to the incapability to express control flow information in DFDs, and the lack of formal semantics (Chen et al., 2005). Test paths are essential for defining test cases since they identify data flow paths from inputs to outputs of the system. On the other hand, the separation of control and data flow supports early functional decomposition, and reduces redundancy among DFDs and corresponding process specifications (DeMarco, 1979).

There are reports that indicate the applicability of DeMarco's Structured Analysis for system testing but do not provide details about the techniques that have been applied. In addition to creating test cases for single processes, these can be successively combined in order to derive system-level test cases (Roper and Bin Ab Rahim, 1993). However, to the best of our experience, the existing approaches do not provide sufficient methodological guidance for deriving system-level tests that consider all data flow paths specified in the DFDs. Furthermore, there is a lack of automatable techniques that support automatic test case derivation based on Structured Analysis. Our approach addresses this gap by enabling automated steps in identifying test paths.

Some authors focus on formal extensions of Structured Analysis, e.g., DFDs enhanced with control flow semantics and conditions (Chen et al., 2005). Functional scenarios, which are related to our understanding of test paths, can be derived based on these formal specifications and then used to guide formal inspections (Li and Liu, 2011). However, these approaches are not applicable to semi-formal modelling languages such as DeMarco DFDs. It requires some subsequent manual validation steps that can notably be performed in reviews. Though both reviews and test case creation are widely recognized requirements validation techniques (Kotonya and Sommerville, 1998), there is little work on combining them, e.g., methodical guidance to support perspective-based reading from the tester's point of view (Shull et al., 2000).

The contribution of this paper is twofold: First, we present a review process model that illustrates how test paths as the basis for test case creation can be used in order to structure and guide requirements reviews. Thereby, we combine testing and reviews as requirements validation techniques in order to complement e.g. perspective-based reading. Second, we propose an automatable technique for exploring test paths based on Structured Analysis (DeMarco, 1979). To this end, we adopt a testability analysis technique for integrated circuits (Robach et al., 1984), and apply it to these semi-formal requirements artifacts. Our technique involves transforming a Structured Analysis specification into an intermediate model, i.e., an Information Transfer Graph (ITG), to facilitate the exploration of system-level test paths. Test path exploration merely considers the syntactic structure of the requirements artifacts to address the lack of formal semantics of the underlying specification languages. This is an advantage for validating early and maybe incomplete requirements. An application example illustrates our approach.

The remainder of this paper is structured as follows: Section 2 summarizes the fundamentals of our approach. Section 3 reviews related work. Section 4 describes how our approach utilizes test paths for guiding requirements reviews. Section 5 presents our technique for exploring test paths in Structured Analysis specifications. Section 6 illustrates the application of our approach. Section 7 concludes the paper.

2 FUNDAMENTALS

2.1 Requirements Validation and Test

Requirements validation aims at checking if a specification succeeds in establishing a common understanding of the system's intentions (Dzida and Freitag, 1998). Requirements reviews or inspections are a commonly used requirements validation technique (Kotonya and Sommerville, 1998). A requirements review is a formal meeting in which a group of stakeholders analyses and discusses requirements documents in order to uncover defects and plan for corrective actions (Kotonya and Sommerville, 1998). Reviews can be guided by Perspective-based Reading, which involves inspecting requirements from different stakeholder's perspectives (Shull et al., 2000).

There is a strong relation between requirements engineering and testing, and a need for better aligning these two disciplines (Graham, 2002). In black-box testing, test inputs and outputs are identified based on the requirements without considering implementation details (Roper, 1994), since the goal is to check if a system's behavior meets its requirements as expected by the stakeholders. Hence, requirements should allow for deriving test cases and defining test criteria that determine if a test is passed or failed (ISO/IEC/IEEE, 2010). The creation of test cases is a useful requirements validation technique, since problems in defining tests indicate, e.g., missing, ambiguous, or incorrect information (Denger and Olsson, 2005). Here, the aim of creating test cases is to validate the requirements rather than a system, as it is in testing (Kotonya and Sommerville, 1998). A test case specifies required test inputs, execution conditions, and expected test results (ISO/IEC/IEEE, 2010). In this paper, we focus on test paths as the basis for defining test cases. In Model-Based Testing, a test path is defined by a sequence of edges that connect initial and final nodes in a graph, as a result of applying a traversal algorithm (Nguyen et al., 2012). Test paths are the basis for abstract test cases, which have to be enhanced by adding specific test input data and expected results (Nguyen et al., 2012).

2.2 Structured Analysis

Structured Analysis provides techniques for analyzing and specifying software requirements. Structured Analysis does not prescribe concrete documentation and modeling languages. In this paper, we focus on the modelling concepts and notations suggested by DeMarco (1979), i.e., Data Flow Diagrams, Data Dictionaries, and Process Specifications.

A Data Flow Diagram (DFD) is a directed graph that consists of three types of nodes: processes, files, and terminators. The nodes are connected via directed edges, i.e., data flows representing the flow of data items between processes. Processes transform incoming data flows into outgoing ones, whereas files serve as repositories for data. Terminators are entities in a system's environment that provide input or receive output data. DFDs do not express control flows, i.e., they neither specify process activation rules nor process sequences. DFDs are used to decompose a system into processes in a hierarchy of abstraction layers. To this end, a process can be refined and described in more detail in a separate DFD.

A Data Dictionary provides a static view; it defines the data elements that are referenced by data flows and files in a DFD. Data definitions specify compositions of atomic or complex data elements using certain operators (e.g., aggregation or multiplicity). A Data Dictionary is especially important for ensuring consistency among hierarchized DFDs, i.e., "balancing" parent and child DFDs.

Process specifications (also called “mini specifications”), as proposed by DeMarco, are written in structured natural language that comprises constructs such as condition or iteration statements, reminiscent of pseudo code. Since DFDs do not specify how process inputs are transformed into outputs, the process logic and control flow of each functional primitive (i.e., low-level process not further decomposed) is described separately in a process specification. Mini specs can be represented in Control Flow Graphs.

2.3 Information Transfer Graph

The Information Transfer Graph (ITG) is a modeling language for analyzing the testability of integrated circuits designs (Robach et al. (1984), and formal data flow design specifications of software (Le Traon and Robach, 1997). In the following, we present the ITG based on (Robach et al., 1984).

An ITG describes the possible flows of data through a system. It can be formally described as a bipartite, directed graph that consists of a set of places and a set of transitions (the graph nodes) as well as a set of edges, i.e., information flows. A place is either a module representing a system component that provides some functionality, a source (input), or a trap (output). By means of transitions, conditions of information transfer are described. There are four different modes of information transfer, as depicted in Fig. 1. For example, the Selection mode means that information is transferred through either one of the outgoing arcs of the first module, while the Distribution mode forwards information to all succeeding modules. Hence, the ITG allows for representing data and control flow (Le Traon and Robach, 1997).

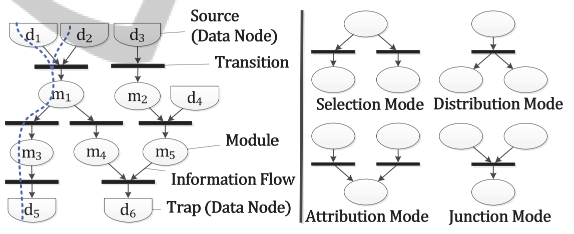

Figure 1: ITG Concepts and Notation.

The modeling concepts described above allow for identifying "flows", i.e., information transfer paths from a certain set of inputs to a set of outputs (Robach et al., 1984). Flows are sub-graphs, each representing an independent function of the system (Robach et al., 1984; Le Traon and Robach, 1997). To identify flows, the semantics of modules and transitions have to be considered. Places constitute “or” nodes while transitions have the semantics of a logical “and”; i.e., a transition requires all incoming information flows to be present in order to proceed in the flow of control (Le Traon and Robach, 1997). Any edge connected to a transition belongs to a flow, while places can be considered as branches and will consequently require separate flows. In Fig. 1, an exemplary flow is highlighted in the ITG.

3 RELATED WORK

Though both reviews and test case creation are widely adopted requirements validation techniques, there are few approaches that combine them. Perspective-based reading from a tester's viewpoint aids reviewers in defining test cases and thereby inspecting requirements specifications (Shull et al., 2000). Test-Case-Driven Inspection (Gorschek and Dzamashvili-Fogelström, 2005) utilizes test cases in requirements reviews, involving testers who create and review them together with stakeholders. In the following, we summarize related work on testing and functional path analysis, focusing on Structured Analysis.

DeMarco (1979) focuses on deriving concrete test inputs for single processes from data definitions in the Data Dictionary, which is also used in Modern Structured Analysis (Yourdon, 1988), a real-time extension. Roper (1994) suggests applying this approach only for small subsets of interacting processes to avoid a large number of infeasible test paths. Roper and Bin Ab Rahim (1993) propose a manual technique for deriving test cases from Structured Analysis specifications. They consider each process in the low-level DFDs separately, and then successively combine these separate test cases to create system-level test cases that consider transformations of system inputs into outputs or data stored in a file. Roper (1994) propose to apply white-box data flow testing techniques on DFDs, based on describing process transformations in terms of consumed and produced data flows.

McCabe and Schulmeyer (1985) propose to derive test cases based on tracing functional dependencies between processes of the DFDs on several abstraction levels to facilitate system and acceptance testing. However, the identification of these dependencies is done manually, and the authors do not provide specific guidance. Emery and Mitchell (1989) describe unit, integration, and system testing approaches based on complexity measures. They propose to identify dependencies between external outputs and inputs by merging low-level DFDs, and creating output-to-input mapping trees.

Other works focus on extensions of DeMarco's Structured Analysis, employing e.g., additional state diagrams (Väliviita et al., 1997). Kan and He (1995) present an approach for deriving Algebraic Petri nets from Modern Structured Analysis specifications, which can be used to formally verify the original specification. In (Chen et al., 2005), an approach for integration testing based on the formalized Condition Data Flow Diagram (CDFD) is proposed. An algorithm for automatically deriving functional scenarios, which relate sets of input and outputs, from CDFDs is presented in (Li and Liu, 2011). Functional scenarios are generated using a depth-first search algorithm, and serve as a guideline for inspections. In the inspection, each functional scenario is reviewed separately, considering each involved operation and its integration (Liu et al., 2010).

In summary, existing approaches do not provide comprehensive methodical guidance for utilizing test artifacts in combination with requirements reviews, as well as for automating the derivation of system-level test cases from DeMarco's Structured Analysis. Regarding formalizations of Structured Analysis, existing work provides test case generation and scenario identification techniques (Chen et al., 2005; Li and Liu, 2011). Though the latter is designed to guide rigorous inspections (Liu et al., 2010) it does not aim at supporting the creation of test cases. Nevertheless, all these formal approaches are not suited to be applied on early DeMarco Structured Analysis specifications due to lack of formal semantics.

4 REQUIREMENTS REVIEWS GUIDED BY TEST PATHS

Our technique for exploring test paths in Structured Analysis artifacts (see Section 5), aims at supporting requirements reviews. We propose to use test paths as guidance for structuring the review process, complementary to perspective-based reading (Shull et al., 2000). Test path exploration can be potentially automated so that the effort and the amount of required testing experience will be significantly reduced, which supports the review team in specifying test cases to validate the specification. In addition to checking requirements correctness, the explored test paths may also uncover additional functionality, or help identify missing requirements.

As this paper focuses on Structured Analysis models, we define a test path as a (logical) sequence of processes and data flows that connect sets of input and output data elements. Thus, test paths determine dependencies between produced outputs and required inputs by orchestrating functions, and are comparable to functional scenarios proposed by Li and Liu (2011). In contrast to these scenarios, we use test paths as templates for abstract test cases without concrete data, which will be analyzed in the review meeting. Our proposed structure of a requirements review based on test paths is depicted in Fig. 2.

Based on the automatically generated test paths, the stakeholders will check if the respective

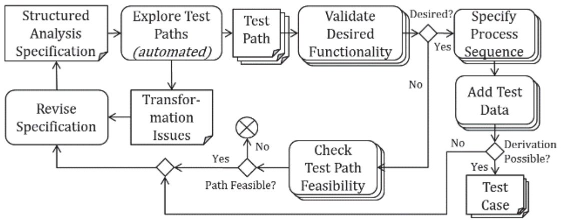

Figure 2: Review Process Supported by Test Paths.

combinations of input and output data flows, as well as the involved processes represent a valid system use case and desired functionality. To this end, they successively take each particular test path as a guideline, and focus on the involved subset of processes that are syntactically related. The stakeholders should first examine the system-level transformation. Then, each process involved in the test path is checked against the stakeholder intentions, so as not to overlook intermediate steps. Process transformations can be recognized in the sets of input and output data of a certain process, as documented in the test path.

If this analysis results in identifying an unintended functionality, it must be checked whether the original specification actually allows for executing that particular test path. Our test path exploration technique (cf. Section 5) merely considers the structure and does not take into account the semantics of the specification. Due to this purely syntactical approach some of the resulting test paths might prove to be infeasible. For instance, branching conditions in the process specifications may prevent certain sequences of processes. However, we argue that the identification and analysis of both feasible and infeasible paths supports the detection of defects in the specification. The stakeholders will review the control flow, as well as the statements and conditions of the involved process specifications to check if an undesired path is feasible. In order to improve readability by simplicity, it is desirable to restructure the specification (e.g., processes and data flows in the DFD, or the control flow of process specifications) so that infeasible test paths are avoided already on a syntactical level. Any syntactical issue discovered during the automatic test path exploration (such as process executions that merely consume inputs) also provides valuable hints for revising the specification.

For each valid and desired test path, the reviewers continue by defining a testing procedure and a related test input sequence in order to specify more concrete system usage scenarios. This is required since the test paths only specify a logical order of exercised processes. Based on these abstract test cases, concrete input test data and expected outputs are added in order to create concrete test cases. This step requires a more thorough review than for feasibility checks, but can again be focused or guided by proceeding along one particular test path at a time. Thereby – besides also inspecting the Data Dictionary – it is assured that each desired test path can be triggered as a test case with appropriate instantiated inputs. Conditional expressions that are in conflict with the feasibility of a desired path will be uncovered by reviewing each involved process specification, and require respective corrections. Furthermore, if the specification lacks information or precision to derive test data, these issues can also be found by creating test cases (cf. Section 2.1). The resulting test cases can later be reused for system and acceptance testing.

5 TEST PATH EXPLORATION

Our approach can be characterized as bottom-up since the starting point is the set of low-level primitive processes. Therefore, we first need to expand all the DFDs into a single DFD containing all the primitive processes (cf. DeMarco, 1979). Based on the control flow graph of each primitive process in the expanded DFD, process transformations (i.e., relations between input and output data flows) can be identified. Similar to Roper and Bin Ab Rahim (1993), we assume that consuming input and producing output data is explicitly indicated by referencing data flows, and traverse the graph w.r.t. the branch coverage criterion. This allows us to focus on the syntactical and structural control flow aspects of the mini specification, and ignore its informally described semantics.

For each resulting transformation path, we document the resulting process transformation as a 2-tuple consisting of the sets of data flows that are consumed and produced by the process, respectively. Note that multiple transformation paths may lead to the same transformation. A transformation is valid if neither the input nor the output data set is empty. Invalid transformations indicate insufficient testability of the mini specification, and are sorted out to be analyzed separately in the validation review (cf. Section 4).

Based on the set of valid transformations of a primitive process, an ITG (cf. Section 2.3) is created for each process separately, which reflects both data flow and control flow aspects of the underlying mini specification. The control flow aspects are implicitly given by the process transformations, each one representing a set of transformation paths, i.e., sequences of statements that carry out this transformation.

An exemplary result of applying our algorithm for creating a process ITG based on a set of process transformations, which is not explained in detail due to space limitations, is illustrated in Fig. 3. A process ITG represents the process transformations extracted from the respective mini specification. The ITG derivation is carried out for each primitive process.

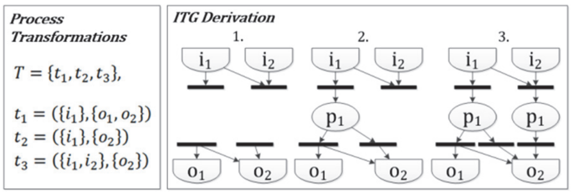

Figure 3: ITG Derivation Example for a Single Process.

Once a separate ITG is derived for each primitive process of the specification, the next step is to combine these ITGs in order to represent the whole system specification. Starting with one arbitrary ITG, all the other ones are successively integrated, which eventually yields the complete ITG. Relations between ITGs can be easily identified by means of data nodes, since an internal data flow is represented by multiple data nodes in different ITGs. This allows for merging two data nodes that are part of two ITGs into one in many cases. In contrast to the literature on the ITG, we also model intermediate data nodes that represent internal data flows between modules.

Furthermore, our approach covers data flows connected to files. Since processes might not write or read a complete record, but only parts of its composition, a further distinction is required. In this case, the composition of the data items in the file is looked up in the Data Dictionary so that data refinements can also be modelled via a transition. In contrast to (Roper and Bin Ab Rahim, 1993), this allows us to identify a more comprehensive set of test paths, which we argue can be useful in validation reviews. Note that internal data nodes might be misinterpreted as system inputs or outputs if, e.g., an atomic data element is stored exclusively while only composite data elements are read from the file (cf. Section 6).

The composite ITG expresses both the data flow and the control flow perspective on the entire system, and is used for test path exploration, i.e., paths that represent transformations performed by the overall system. To this end, we utilize the concept of "flows", i.e., sub-graphs of an ITG (cf. Section 2.3). For each output data node, graph traversal is applied backwards (i.e., in the opposite direction of information flow), by taking into account the different semantics of places and transitions. Multiple incoming information flow edges of a module or data node are handled as disjoint branches. Incoming and outgoing edges of a transition are traversed concurrently, i.e., all these connected edges will eventually be part of the respective flows (Robach et al., 1984). In the case of multiple outgoing edges of a transition, additional forward traversal is applied in order to identify the connected output data nodes. Thus, our approach identifies sets of related in- and outputs, in contrast to (Emery and Mitchell, 1989). This may result in identifying the same flow based on different traversal starting points so that redundant flows must be eliminated. According to Le Traon and Robach (1995), a loop in the ITG will be represented by two flows; one flow exercises the loop while the other passes it by.

Since we aim at facilitating the validation of a Structured Analysis specification, test paths should be represented independently of the ITG, which is only an intermediate artifact. We suggest using a table in which the logical order of the involved processes is expressed (see example in Table 1 below). In contrast to scenarios, test paths constitute the basis for test case creation (cf. Section 4), and neither express temporal sequences of functions nor specific sequences of providing input data to the system.

6 APPLICATION EXAMPLE

For initial evaluation, we applied our technique for exploring test paths described in Section 5 to a simple example, i.e., a fictional Structured Analysis specification of a library information system (LIS). The LIS specification comprises three levels of abstraction, i.e., the context diagram is decomposed into four system processes (level 0), one of which is further decomposed. Fig. 4 shows the expanded DFD.

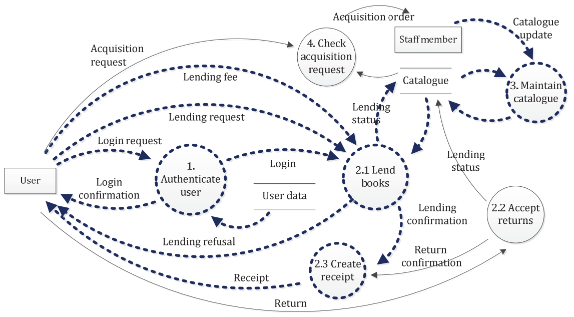

Figure 4: Expanded DFD of the LIS Example.

Fig. 5 shows the ITG that represents the entire system as a result of combining all the ITGs for the six primitive processes shown in the expanded DFD in Fig. 4.

An exemplary flow is highlighted, which has been identified via flow analysis.

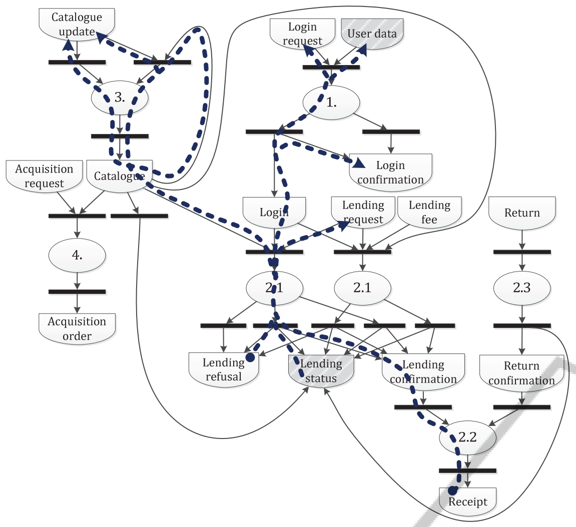

Figure 5: Complete ITG and Exemplary Flow.

This exemplary flow constitutes one potential test path as a result of applying our technique. Three different output data nodes (i.e., *Lending refusal*, *Lending status* and *Receipt*) can be the starting point for identifying this flow. Note that *Lending status* is actually no output data node, since it can be traced back to a data flow that saves information to a file, which is never read exclusively. Similarly, *User data* actually represents a data flow read from a file, since the corresponding file does not have any incoming data flows. In Table 1 we show how the exemplary test path can be documented. External inputs and outputs, i.e., data flows between the system and its context, are distinguished from internal ones. In Fig. 4 this exemplary test path is highlighted in the expanded DFD.

In the subsequent review supported by test paths (cf. Section 4), the exemplary test path will be manually reviewed by stakeholders. Focusing first on the sets of system inputs and outputs, the reviewers will find that the respective system functionality represents a valid and desired use case. Consequently, they will continue with a more thorough review to define a concrete process sequence and concrete test data to invoke this system behavior and observe expected outputs. Here, a set of books that constitute a *Lending request* are evaluated. To check their lending status (e.g., "reserved"), the *Catalogue* file is read. The set of books includes both lendable and not lendable books, so that at least one *Lending refusal*, as well as some *Receipt* for a successful lending is created. The user also has to be authenticated, which requires a *Login request* as well as reading the *User data* file. The test path also involves the process of maintaining the catalogue, since books need to be recorded in the library's *Catalogue* file before they are available for lending. This process is exercised twice in a loop, meaning that the catalogue is empty first, so that subsequent updates involve reading the file. To validate this scenario, the reviewers will review each involved process specification in order to specify a real test case that covers this system behavior.

7 DISCUSSION

In this paper, we proposed an approach that fosters the early requirements validation of Structured Analysis specifications by combining testing and review techniques. We sketched an automatable technique for exploring test paths in Structured Analysis specifications consisting of different artifacts on several abstraction layers. We especially address the gap of support for creating test cases based on these semi-formal specifications. Test paths are the basis for defining (abstract) test cases, and can be used for guiding and structuring specification reviews to uncover incorrect requirements. Our test path exploration technique is especially suited for early requirements validation since it merely considers the syntactical structure of the artifacts and does not take into account their informally specified semantics.

Our overall approach is scalable as it complements manual review processes with automated test path exploration, and supports time-consuming test case creation. It is particularly difficult to manually identify dependencies between global in- and outputs across many DFD levels manually. The test path exploration algorithm involves some steps whose computation can be simplified. For instance, the complexity of identifying process transformations in mini specifications with iterations can be reduced by

Table 1: Exemplary Test Path.

Order	Process	Internal inputs	Internal outputs	System inputs	System outputs
1	3	*Ø*	*{Catalogue}*	*{Catalogue update}*	*Ø*
2	3	*{Catalogue}*	*{Catalogue}*	*{Catalogue update}*	*Ø*
	1	*{User data}*	*{Login}*	*{Login request, User data}*	*{Login confirmation}*
3	2.1	*{Catalogue, Login}*	*{Lending confirmation, Lending status}*	*{Lending request}*	*{Lending status, Lending refusal}*
4	2.2	*{Lending confirmation}*	*Ø*	*Ø*	*{Receipt}*

considering loop bodies separately, since the focus is on the structural aspects. Transformation paths found in loop bodies can be successively combined with the ones identified from the superstructure in order to avoid path coverage issues.

Though our test path exploration technique particularly suits data flow oriented Structured Analysis, it can, in principle, be applied to other modeling languages, such as UML Activity Diagrams or BPMN. To cover approaches that merely focus on control flow, using the ITG as a simple model to explicitly specify the transfer of data may be an option.

REFERENCES

Boehm, B., Basili, V.R., 2001. Software Defect Reduction Top 10 List. In *IEEE Computer*, vol. 34, no. 1, pp. 135-137. IEEE.

Chen, Y., Liu, S., Nagoya, F., 2005. An Approach to Integration Testing Based on Data Flow Specifications. In *1st Int. Colloquium Theoretical Aspects of Computing*, LNCS 3407, pp. 235-249. Springer.

DeMarco, T., 1979. *Structured Analysis and System Specification*, Prentice-Hall. Englewood Cliffs.

Denger, C., Olsson, T., 2005. Quality Assurance in Requirements Engineering. In *Engineering and Managing Software Requirements*, pp. 163-185. Springer.

Dzida, W., Freitag, R., 1998. Making Use of Scenarios for Validating Analysis and Design. In *IEEE Trans. Softw. Eng.*, vol. 24, pp. 1182-1196. IEEE.

Emery, K.O., Mitchell, B.K., 1989. Multi-Level Software Testing Based on Cyclomatic Complexity. In *Proc. Nat. Aerospace and Electronics Conf.*, pp. 500-507. IEEE.

Giaglis, G.M., 2001. A Taxonomy of Business Process Modeling and Information Systems Modeling Techniques. In *Int. J. Flexible Manufacturing Systems*, vol. 13, pp. 209-228. Springer.

Gorschek, T., Dzamashvili-Fogelström, N., 2005. Test-case Driven Inspection of Pre-project Requirements – Process Proposal and Industry Experience Report. In *Requirements Engineering Decision Support Workshop*.

Graham, D., 2002. Requirements and Testing – Seven Missing-Link Myths. In *IEEE Softw.*, vol. 19, issue 5, pp. 15-17. IEEE.

IIBA, 2009. *A Guide to the Business Analysis Body of Knowledge (BABOK® Guide)*, International Institute of Business Analysis. Toronto.

IREB, 2012. *Syllabus IREB Certified Professional for Requirements Engineering – Foundation Level*, Version 2.1, Int. Requirements Engineering Board.

ISO/IEC/IEEE, 2010. *Systems and software engineering – Vocabulary*, ISO/IEC/IEEE 24765, First Edition.

Kan, C.-Y., He, X., 1995. Deriving Algebraic Petri Net Specifications from Structured Analysis – A Case Study. In *Information and Software Technology*, vol. 37, issue 8, pp. 411-434. Elsevier.

Kotonya, G., Sommerville, I., 1998. *Requirements Engineering – Processes and Techniques*, John Wiley & Sons. Chichester.

Le Traon, Y., Robach, C., 1995. Towards a Unified Approach to the Testability of Co-Designed Systems. In *Proc. 6th Int. Symp. Software Reliability Engineering*, pp. 278-285. IEEE.

Le Traon, Y., Robach, C., 1997. Testability Measurements for Data Flow Designs. In *Proc. 4th Int. Software Metrics Symp.*, pp. 91-98. IEEE.

Li, M., Liu, S., 2011. Automatically Generating Functional Scenarios from SOFL CDFD for Specification Inspection. In *Proc. IASTED Int. Conf. Software Engineering*, pp. 18-25. ACTA Press.

Liu, S., McDermid, J.A., Chen, Y., 2010. A Rigorous Method for Inspection of Model-Based Formal Specifications. In *IEEE Trans. Reliab.*, vol. 59, issue 4, pp. 667-684. IEEE.

McCabe, T.J., Schulmeyer, G.G., 1985. System Testing Aided by Structured Analysis – A Practical Experience. In *IEEE Trans. Softw. Eng.*, vol. SE-11, pp. 917-921. IEEE.

Nguyen, C.D., Marchetto, A., Tonella, P., 2012. Combining Model-Based and Combinatorial Testing for Effective Test Case Generation. In *Proc. 2012 Int. Symp. Software Testing and Analysis*, pp. 100-110. ACM.

Pressman, R.S., 2010. *Software Engineering – A Practitioner's Approach*, McGraw-Hill. New York, 7th ed.

Robach, C., Malecha, P., Michel, G., 1984. CATA – A Computer-Aided Test Analysis System. In *IEEE Des. Test Comp.*, vol. 1, pp. 68-79. IEEE.

Roper, M., 1994. *Software Testing*, International Software Quality Assurance Series, McGraw-Hill. London.

Roper, M., Bin Ab Rahim, A.R., 1993. Software Testing Using Analysis and Design Techniques. In *Software Testing, Verification and Reliability*, vol. 3, issue 3-4, pp. 165-179. Elsevier.

Shull, F., Rus, I., Basili, V., 2000. How Perspective-Based Reading can Improve Requirements Inspections. In *IEEE Computer*, vol. 33, issue 7, pp. 73-79. IEEE.

Uusitalo, E.J., Komssi, M., Kauppinen, M., Davis, A.M., 2008. Linking Requirements and Testing in Practice. In *16th IEEE Int. Requirements Engineering Conf.*, pp. 265-270. IEEE.

Väliviita, S., Tiitinen, P., Ovaska, S.J., 1997. Improving the Reusability of Frequency Converter Software by Using the Structured Analysis Method. In *Proc. IEEE Int. Symp. Industrial Electronics*, vol. 2, pp. 229-234. IEEE.

Yourdon, E., 1988. *Modern Structured Analysis*, Prentice-Hall. Englewood Cliffs.