FORMAL SOFTWARE INSPECTIONS: AN INDUSTRIAL APPLICATION OF FUNCTION TABLES AND EVENT-B TO SOFTWARE OF A WAYSIDE TRAIN MONITORING SYSTEM

ROBERT ESCHBACH
ITK Engineering GmbH, Germamy

ABSTRACT

The experience gained in the industrial application of software inspections using Function Tables and Event-B to a subsystem of a Wayside Train Monitoring system (WTMS) is presented in this paper. The WTMS Configuration Management System (CMS) supports the creation and management of configuration data for the WTMS. The correct and reliable implementation of the required system functions, especially those dealing with data handling and data management, is of particular importance for the overall quality of the system since faults in these functions may lead to critical failures and malfunctioning. Therefore, the development of the data handling part of a CMS requires the use of high integrity methods like systematic software inspections in order to ensure the highest quality. Function Tables have been successfully applied for the inspection of safety-critical software. In our industrial project, a special variant of Function Tables was defined that can be easily mapped to formal Event-B specifications. Event-B with its set-theoretic basis for modeling, its concept of refinement and the use of formal proof to ensure correctness of refinement steps, is used to formally analyze the derived Function Tables. The systematic derivation of Function Tables is done by a verification-based inspection using reading technique "stepwise abstraction".

Keywords: software inspection, Function Table, Event-B, stepwise abstraction.

1 INTRODUCTION

The purpose of a Wayside Train Monitoring System (WTMS) is to detect early threats that may lead to hazards and damages by monitoring trains and environmental conditions [1]. Typical examples are the detection of hot box, brake-locking or load displacements. A WTMS is a highly distributed system that, from a system data perspective, gathers, analyzes and exchanges different kinds of measurement and status data. A central part of a WTMS is the Configuration Management System (CMS). It is responsible for the correct data exchange between connected devices and control and management service points. The goal of our project was the development of a server-based subsystem of the CMS. In Eschbach [2] the formal conceptual data analysis of CMS management data using Event-B has been presented. In this paper the experiences gained in the systematics inspection of software using Function Tables and Event-B will be described.

In Basili and Selby [3] the authors compared three software quality assurance techniques: (a) code reading by stepwise abstraction, (b) functional testing using equivalence partitioning and boundary value analysis, and (c) structural testing using 100% statement coverage criteria. These techniques were compared w.r.t. three aspects: fault detection effectiveness, fault detection cost, and classes of faults detected. One major outcome of this study is summarized by the authors as:

> With the professional programmers, code reading detected more software faults and had a higher fault detection rate than did functional or structural testing, while functional testing detected more faults than did structural testing, but functional and structural testing were not different in fault detection rate.

WIT Transactions on The Built Environment, Vol 199, © 2020 WIT Press
www.witpress.com, ISSN 1743-3509 (on-line)
doi:10.2495/CR200271

In Linger et al. [4] this reading technique is characterized as follows:

> The process of reading a poorly documented program bottom up is called *stepwise abstraction*. Stepwise abstraction may be required in either the verification of correctness or the determination of program function.

Based on these results we chosen the reading technique stepwise abstraction for software inspection in our project. The applied inspection technique was strongly influenced by the cleanroom inspection methods (see Mills et al. [5] and Mills [6]). The cleanroom inspection methods are verification-based methods with the goal of finding functional defects in software through mathematical verification building upon the box-structured method (see Mills [7]). In our project we have used Function Tables in order to abstract from concrete clear-box software structures to more abstract state box and black box structures.

The results of Elberzhager et al. [8] were used to integrate the software inspection with our testing processes. For this purpose, project-specific assumptions have been identified and contextfactors have been derived in order to integrate our inspection processes based on stepwise abstraction with our testing processes. Furthermore, so called Goal Indicator Trees (GITs), as defined in Kloos et al. [9], which support inspectors in performing an inspection, thereby assuring the quality of work products and improving the quality of the overall system, were used to identify the relevant quality objectives and refine them into appropriate quality indicators. One outcome of this analysis was to focus formally on software functions that may cause database inconsistencies.

In Powell [10] the authors present a verification-based software inspection technique using tool support. The proposed tool extracts and generates proof obligations directly from software and supports and guides an inspector by his semi-formal verification. The proposed approach does not lead to a formal model of software functions as intended in our project. How an inspection process can be improved by focusing and tailoring the inspection on experiences and expertise of the reviewers is described in Parnas and Weiss [11]. The proposed approach does not directly address formal specification and verification but is of great practical value. In Singh et al. [12] the authors present a refinement strategy to generate formal Event-B models from tabular expressions. The refinement strategy, especially the generation of models is based on a correct-by-construction approach. The proposed approach will be illustrated by an insulin infusion pump case study. The approach addresses formal specification and verification but is not used for formal software inspection, which requires an appropriate abstraction strategy. Module interface specification enriched with assertions and semi-formal proofs can be of great advantage, when inspecting code w.r.t. contracts given by pre- and postconditions. These results are described in Jackson and Hoffman [13]. The paper covers some of the underlying ideas of our verification-based software inspections technique, but does not address suitable abstraction strategies as well as the formal specification and verification aspects. The verification-based inspection technique presented in this paper is based on a systematic abstraction strategy that allows for stepwise abstraction and the creation of a formal Event-B model. Furthermore, the presented inspection technique is scalable w.r.t. the degree of formality, i.e. depending on the concrete context it is possible to work with an informal model specified with natural language statements as well as with a complete formal Event-B model. Techniques like the GITs, as defined in Kloos et al. [9], can be used to derive criteria, that supports the decision which degree of formality is appropriate in a concrete context.

Event-B [14], [15] with its set-theoretic basis for modeling, its concept of refinement and the use of formal proofs ensuring correctness of refinement steps has been used for the formal code inspection based on Function Tables and reading technique stepwise abstraction.

Formal methods, especially Event-B have been applied successfully in many safety-critical railway systems [16]–[18]. The success of Event-B and the B method has even an influence on the definition of CENELEC standard EN50128 [19]. A state-of-the-art of methods and tools for the verification of interlocking systems can be found in Haxthausen and Peleska [20].

In the following it will be described how the application of Function Tables and Event-B together with the reading technique stepwise abstraction can be successfully used to systematically inspect and analyze critical software.

2 WAYSIDE TRAIN MONITORING SYSTEM (WTMS)

2.1 Architecture

The CMS can be used to create configuration data for the WTMS. On an abstract level the CMS consists of data and operations related to this data (see Fig. 1). CMS data can be further divided into data for the ticket system and technical configuration data related to the WTMS. The CDM and FCDM has been created for the ticket system.

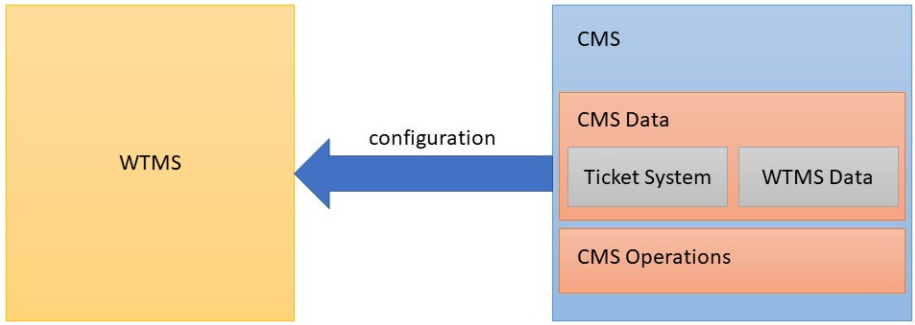

Figure 1: Conceptual view of WTMS and CMS.

2.2 Challenges and solutions

Since a CMS is responsible for correct data exchange, data integrity is a very important quality attribute. Furthermore, a systematic handling of faults is very important. From an engineering point of view the main focus was on fault prevention rather then fault identification and removal. Another important concept is fault tolerance. Strategies for fault tolerance help to avoid failures during system execution by means of error detection and recovery mechanisms. Using database constraints and triggers based on formal models as well as SQL statements involved in transactions with roll-back mechanisms, a specific kind of fault tolerance has been achieved.

Data integrity and especially data consistency of system data is one of the most important quality properties of a CMS since inconsistency may lead to critical malfunctioning. Therefore, the development of a CMS requires the use of high integrity methods in order to ensure the highest quality.

During system development, it is important to establish techniques for *fault prevention*. A fault may cause errors during system execution and may lead to critical situations. Different techniques for fault prevention exist. For example, coding guidelines or systematic code

inspections may help software developers to prevent faults. The overall goal should be to prevent almost all faults in order to deliver (almost) zero-fault software.

Another important quality attribute required for the CMS is *fault tolerance*. Since each non-trivial system may have faults, it is very important to deal with faults in a systematic way. Fault tolerance covers different techniques for error detection as well as for system recovery. A special way to recover a consistent system state in a CMS is to remove inconsistent data and roll back to an earlier consistent and saved data restoration point. In our project, a sequence of database statements was embedded into a transaction with associated rollback mechanisms. Whenever an exception occurs (for example, by violating a unique or foreign key constraint), the database will roll back to the last saved restoration point. The function *ewt* will check whether the execution of the transaction was successful or not and will trigger a warning in the latter case.

3 EVENT-B

Event-B is a formal method that can be used to model and analyze state transition systems (see Abrial [14]). Event-B is part of the B-Method (see Abrial [15]). Event-B supports refinement-based development of transition systems. It is based on of first-order logic and typed set theory. Tool support for Event-B is given by the Rodin Platform (http://www.event-b.org). An Event-B specification consists of a static and a dynamic part. The static part describes constants (context), the dynamic part behavior (machine).

A context is used for specifying sets, constants, axioms and theorems. Axioms and theorems are used to state properties of sets and constants. In this paper execution modes were specified as constants among others (see Fig. 2). The context *ctx − ewt* depicted in Fig. 2 defines a set *ExecMode*, four constants *start*, *execute*, *wait* and *exit*. The axiom states that *ExecMode* = {*start,execute,wait,exit*} and that set *ExecMode* has exactly four elements.

```
context
    ctx−ewt
sets
   ExecMode
    . . .
constants
   start
    execute
    wait
    exit
    . . .
axioms
   partition (ExecMode , { start } , { execute } , { wait } , { exit })
    . . .
end
```

Figure 2: Context ewt.

A machine describes transitions (events) and may use the information of a connected context. A machine specifies variables, invariants and events. An event describes a transition and consists of a possibly empty set of event parameters, guards, that describes conditions

under which an event is enabled and actions, that describe how parameters will change, if the event occurs (see Fig. 3).

```
machine
    ewt
sees
    ctx−ewt
variables
    phase
    current_db_state
    consistent_db_state
    transaction
    . . .
invariants
    phase ∈ {ewt , ef } → { start , execute , exit , wait }
    phase ( ewt ) = e x i t ⇒
        consistent_db_state ( current_db_state) = TRUE
    . . .
end
```

Figure 3: Machine ewt.

The machine *ewt* depicted in 3 can use context *ctx − ewt* (sees clause) and defines the variables *phase*, *current_db_state*, *consistent_db_state* and *transaction*. The invariant *phase* ∈{*ewt,ef*}→{*start,execute,exit,wait*} states that *phase* is a total function with domain {*ewt,ef*} and range {*start,execute,exit,wait*}. The second invariant states, that when *phase*(*ewt*) = *exit* holds, the current db state *current_db_state* is consistent.

4 SOFTWARE INSPECTIONS

Since data consistency is one of the most important quality attributes of the CMS, the decision was made to systematically inspect almost all software providing functionalities with database access. The reading technique stepwise abstraction was chosen based on the results given in Basili and Selby [3].

In our project, the execution of a sequence of database statements are always embedded into a SQL transaction with rollback mechanism. A simplified skeleton of the function that embeds the execution of multiple SQL-instructions into a transaction is shown in Fig. 4.

Function *ewt* has function pointer *ef* as argument. In the first step, relevant program variables of *ewt* will be identified. In this example these are the following variables:

1. *rid* (return value of function *ewt*),
2. *rc* (assigned to the return value of function *ef*).

We use so-called meta-variables for expressing abstract properties. The status of a transaction will be abstractly represented by meta variable *transaction*

$$transaction \in \{tr\_begin, tr\_commit, tr\_rollback\}. \quad (1)$$

If function *ewt* starts a transaction, we simply write

$$transaction := tr\_begin. \quad (2)$$

```
int ewt ( const std : : function <bool ()> &ef ) {
    try {
        tr_begin ( db ) ;
        bool rc = ef ( ) ;
        if ( ! rc ) { throw Exc ( " ef␣ . . . " ) ; }
        tr_commit ( db ) ;
        rid = . . . ;
        return rid ; }
    catch ( const C_Exc &exc ) {
        tr_rollback ( db ) ;
        throw ; } . . . }
```

Figure 4: C++-Function ewt.

In the same way, a transaction commit or rollback wil be treated. Since function *ef* may throw an exception, we introduce a meta variable *exception*, which is either *TRUE* or *FALSE*. The following condition describes, that *ef* has thrown an exception:

$$exception = TRUE. \tag{3}$$

Furthermore, we define a meta variable for the current database state *current_db_state* and the current restoration point *restoration_point*. Since we just want to distinguish database states, we model both meta variables as natural numbers, i.e.

$$current\_db\_state, restoration\_point \in \mathbb{N}. \tag{4}$$

The consistency of db states is represented by meta variable *consistent*, which is either *TRUE* or *FALSE*. The execution phases of functions *ewt* and *ef* will be represented by variable

$$phase : \{ewt, ef\} \rightarrow \{start, execute, wait, exit\}. \tag{5}$$

In phase *start* variables and meta variables will be initialized. Phase *execute* covers the execution and phase *wait* is used to model the exection of other Function Tables. The execution ends in phase *exit*.

During the stepwise construction of Function Table FT-EWT (see eqn (5)), the explicit scheduling of function *ewt* will be replaced by an implicit scheduling based on conditions. The table is created bottom up using the reading technique stepwise abstraction as defined in Linger et al. [4]. The Function Tables presented in this paper are a variant of those proposed by Dave Parnas (see [21], for example).

Each column of the derived Function Table, as shown in Fig. 5, describes a specific condition and associated actions, i.e. a parallel assignment of values to variables. In our example, a condition is related to:

- the current execution phase $phase \in \{start, execute, wait, exit\}$,
- the existence of an exception $exception \in \{TRUE, FALSE\}$ and
- the return code $rc \in \{undef, ok, nok\}$.

Initially, the execution starts with $phase(ewt) = start$ and $phase(ef) = wait$. The conditions are organized hierarchically. The Function Table can be read column by column starting with the column condition, then going to the values that will be assigned to the specified variables

Event Name	ewt-start	ewt-execute-1	ewt-execute-2	ewt-execute-3	ewt-execute-4	ewt-exit
Conditions / Variables	phase =					
	start	execute				exit
		exception =				
		FALSE			TRUE	
		rc =				
		undef	ok	nok		
rid	-1/ok	nc	last_insert_rowid/ok	nc	nc	nc
rc	undef	ef()	nc	nc	nc	nc
transaction	undef	tr_begin	tr_commit	nc	tr_rollback	nc
exception	FALSE	nc	undef	TRUE	nc	nc
restoration_point	undef	current_db_state	undef	nc	undef	nc
current_db_state	nc	nc	new_db_state	new_db_state	restoration_point	nc
consistent	nc	nc	TRUE	FALSE	nc	nc
phase	execute	nc	exit	nc	exit	nc

Values

Figure 5: Function Table FT-EWT.

in parallel. A value cell marked with *nc* (no change) does not cause a change of the corresponding variable.

Each column of this table is associated with an event in Event-B with guards related to the column conditions and actions defined as parallel assignment of the column values to variables. For example, event *ewt-execute-1*, as shown in Fig. 6, has been derived from column with *Event Name = ewt-execute-1*. The column condition is

$$phase(ewt) = execute \wedge exception = FALSE \wedge rc = undef \wedge ef \in \{ok,nok\}. \quad (6)$$

The parallel assignment can be specified as

$$rc := ef,transaction := tr\_begin,restoration\_point := current\_db\_state. \quad (7)$$

The corresponding event defined in Event-B can be seen in Fig. 6.

```
Ewt_execute_1
any
    ef
where
    phase ( ewt ) = execute
    exception = FALSE
    rc = undef
    ef ∈ {ok , nok}
then
    rc := ef
    transaction := tr_begin
    restoration_point := current_db_state
end
```

Figure 6: Event ewt_execute_1.

The main objective of the inspection was to verify that in phase *exit* the current data base state is consistent. This can be formalized as following invariant:

$$phase(ewt) = exit \Rightarrow consistent(current\_db\_state) = TRUE. \quad (8)$$

The formal analysis was performed with the eclipse-based tool Rodin. The verification of all invariants, especially invariant 8, was performed with theorem provers, SMT solvers available in Rodin (using appropriate plugins). Furthermore, the model checker ProB was also used to prove the deadlock freeness of the model. The state space has been explored by stepwise execution of the model. The inspection of *ewt* und four instances of *ef* revealed several serious incompleteness faults.

5 CONCLUSION

The knowledge gained in the application of systematic software inspections based on Function Tables and Event-B to the configuration management system of a wayside train monitoring system has been presented in this paper. The configuration management system supports the creation and management of configuration data for the wayside train monitoring system. The correctness and reliability of all software functions dealing with data handling and data management is of utmost importance. Systematic software inspections of these functions have been successfully used for fault detection. The reading technique stepwise abstraction was used to create Function Tables that were later on mapped to Event-B specifications for critical functions. In this way a formal analysis based on several formal verification techniques like theorem proving, SMT solving, model checking and constraint solving was possible. Furthermore animation of the functional behavior was used for validation purposes. Several critical incompleteness faults could be identified and corrected. The Function Tables, especially informal tables, were of great practical value since they were used not only for fault detection but also for creating common understanding within the development team. The reading technique stepwise abstractions helps to get the analyzed functionalities under intellectual control. Event-B as underlying formal method has greatly improved the analysis of critical functions. Especially the integration of formal specifications and formal verification, as well as the refinement concept, makes Event-B to a practical and powerful formal technique useful in industrial projects.

REFERENCES

[1] Bracciali, A., Wayside train monitoring systems: A state-of-the-art and running safety implications. *International Journal of Railway Technology*, **1**(1), pp. 231–247, 2012.

[2] Eschbach, R., Industrial application of Event-B to a wayside train monitoring system: Formal conceptual data analysis. *Formal Methods – The Next 30 Years*, eds M.H. ter Beek, A. McIver & J.N. Oliveira, Springer International Publishing, pp. 738–745, 2019.

[3] Basili, V. & Selby, R., Comparing the effectiveness of software testing strategies. *IEEE Transactions on Software Engineering*, **SE-13**(12), pp. 1278–1296, 1987.

[4] Linger, R.C., Mills, H.D. & Witt, B.I., *Structured Programming, Theory and Practice*, The Systems Programming Series, Addison-Wesley, 1979.

[5] Mills, H., Dyer, M. & Linger, R., Cleanroom software engineering. *IEEE Software*, **4**(5), pp. 19–25, 1987.

[6] Mills, H.D., Zero defect software: Cleanroom engineering. *Advances in Computing*, **36**, pp. 1–41, 1993.

[7] Mills, H., Stepwise refinement and verification in box-structured systems. *Computer*, **21**(6), pp. 23–36, 1988.

[8] Elberzhager, F., Eschbach, R. & Munch, J., The relevance of assumptions and context factors for the integration of inspections and testing. *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, IEEE, pp. 388–391, 2011.

[9] Kloos, J., Elberzhager, F. & Eschbach, R., Systematic construction of goal indicator trees for indicator-based dependability inspections. *2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications*, IEEE, pp. 279–282, 2010.
[10] Powell, D., Tool support for verification-based software inspection. *2004 Australian Software Engineering Conference Proceedings*, IEEE, pp. 232–240, 2004.
[11] Parnas, D. & Weiss, D., Active design reviews: Principles and practices. *Journal of Systems and Software*, **7**(4), pp. 259–265, 1987.
[12] Singh, N.K., Lawford, M., Maibaum, T.S.E. & Wassyng, A., Use of tabular expressions for refinement automation. *Model and Data Engineering*, eds. Y. Ouhammou, M. Ivanovic, A. Abello & L. Bellatreche, Springer International Publishing, Vol. 10563 of *Lecture Notes in Computer Science*, pp. 167–182, 2017.
[13] Jackson, A. & Hoffman, D., Inspecting module interface specifications. *Software Testing, Verification and Reliability*, **4**(2), pp. 101–117, 1994.
[14] Abrial, J.R., *Modeling in Event-B: System and Software Engineering*, Cambridge University Press: Cambridge, 2010.
[15] Abrial, J.R., *The B-Book: Assigning Programs to Meanings*, Cambridge University Press: Cambridge, 1996.
[16] Lecomte, T., Servat, T. & Pouzancre, G., Formal methods in safety-critical railway systems. *10th Brazilian Symposium on Formal Methods*, 2007.
[17] ter Beek, M.H., Fantechi, A., Ferrari, A., Gnesi, S. & Scopigno, R., Formal methods for the railway sector. *ERCIM News*, **112**, 2018.
[18] Ferrari, A. et al., Survey on formal methods and tools in railways: The astrail approach. *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification*, eds S. Collart-Dutilleul, T. Lecomte & A. Romanovsky, Springer International Publishing: Lille, pp. 226–241, 2019.
[19] Fantechi, A., Fokkink, W. & Morzenti, A., *Some Trends in Formal Methods Applications to Railway Signaling*, John Wiley & Sons: Hoboken, pp. 61–84, 2012.
[20] Haxthausen, A.E. & Peleska, J., *Model Checking and Model-Based Testing in the Railway Domain*, Springer Fachmedien Wiesbaden, pp. 82–121, 2015.
[21] Parnas, D.L., Inspection of safety-critical software using program-function tables. *Proceedings of the IFIP 13th World Computer Congress, Hamburg, Germany,* pp. 270–277, 1994.