Enhanced V-Model

Article *in* Informatica · April 2018

DOI: 10.31449/inf.v42i4.2027

CITATIONS
7

READS
5,152

4 authors, including:

Mustafa Seçkin Durmuş
Sadenco Engineering
48 PUBLICATIONS **291** CITATIONS
SEE PROFILE

Ilker Ustoglu
Yildiz Technical University
22 PUBLICATIONS **61** CITATIONS
SEE PROFILE

Roman Tsarev
Siberian Federal University
102 PUBLICATIONS **361** CITATIONS
SEE PROFILE

https://doi.org/10.31449/inf.v42i4.2027 Informatica **42** (2018) 577–585 **577**

Enhanced V-Model

Mustafa Seçkin Durmuş
Sadenco, Safe, Dependable Engineering and Consultancy, Antalya, Turkey
E-mail: msd@saden.co

İlker Üstoğlu
Yildiz Technical University, Department of Control and Automation Engineering, Istanbul, Turkey
E-mail: ustoglu@yildiz.edu.tr

Roman Yu. Tsarev
Siberian Federal University, Department of Informatics, Krasnoyarsk, Russia
E-mail: tsarev.sfu@mail.ru

Josef Börcsök
Kassel University, Computer Architecture and System Programming Department, Kassel, Germany
E-mail: j.boercsoek@uni-kassel.de

Keywords: Software development lifecycle, V-model, fault diagnosis, discrete event systems, EN 50128, fixed-block railway signaling systems

Received: November 16, 2017

Typically, software development processes are time consuming, expensive, and rigorous, particularly for safety-critical applications. Even if guidelines and recommendations are defined by sector-specific functional safety standards, development process may not be completed because of excessive costs or insufficient planning. The V-model is one of the most well-known software development lifecycle model. In this study, the V-model lifecycle is modified by adding an intermediate step. The proposed modification is realized by checking the fault diagnosability of each module. The proposed modification provides three advantages: (1) it checks whether the constructed model covers all software requirements related with faults; (2) it decreases costs by early detection of modeling deficiencies before the coding and testing phases; and (3) it enables code simplicity in decision of fault occurrence.

Povzetek: Osnovnemu modelu V razvoja programskih sistemov je dodana izboljšava na osnovi možnosti testiranja napak modulov.

1 Introduction

The concept known as Safety Integrity Level (SIL) is used to quantify safety. The SIL is a degree of safety system performance for a Safety Instrumented System (SIS), which is an automatic system used to avoid accidents and to reduce their impact both on humans and the environment. A SIS has to execute one or more Safety Instrumented Functions (SIFs) to maintain a safe state for the equipment under control [1]. Bear in mind that, a safe state is known as the state where the whole system is prevented from falling into a dangerous situation. A SIF has a designated SIL level depending on the ratio of risk that needs to be decreased. IEC 61508, the standard for functional safety of electrical/electronic/programmable-electronic Safety Related System (SRSs), mentions that a SIL should be designated to each SIF and defines the safety integrity as the probability of a SRS adequately performing the required safety functions under all the stated conditions within a given period of time from the lowest requirement level (SIL 1) to highest requirement level (SIL 4).

The third part of IEC 61508 applies to any software used to develop a safety-related system within the scope of first and second parts of the standard, and establishes the requirements for *safety lifecycle* phases. Industry and domain specific implementations of IEC 61508 include IEC 61511 for industrial processes, IEC 61513 for the nuclear industry, and IEC 62061 for machinery etc.

A lifecycle model is defined in [2] as a *model that describes stages in a software product development process*. The IEC 61508-4 standard discusses the term lifecycle in the context of both *safety lifecycle* and *software lifecycle*. The safety lifecycle includes the necessary activities involved in the implementation of SRSs [3]. IEC 61508 states that a *safety lifecycle for software development shall be selected* and specified during the safety-planning phase in accordance with Clause 6 of IEC 61508-1. The safety lifecycle includes the definition of scope, hazard and risk analysis, determination of safety requirements, installation, commissioning, validation, operation, maintenance, repair, and decommissioning. On the other hand, the software lifecycle includes the activities occurring from the conception of the software to the decommissioning of the software.

Numerous lifecycle models have been addressed in the literature, such as the waterfall, spiral, iterative development, and butterfly models [4-8]. However, despite the availability of many lifecycle alternatives, safety standards such as IEC 61508, EN 50126, EN 50128, and IEC 62278 recommend using the V-model for software development processes. The V-model lifecycle has been applied to various domains such as the automotive [9], aerospace [10], railways [11], and the nuclear industry [12].

In this study, a Discrete Event System (DES)-based fault diagnosis method is added to the V-model lifecycle as an intermediate step between the module design and the coding phases. A DES is a discrete-state, event-driven system in which the state evolution of the system depends totally on the occurrence of discrete events over time.

The main difference of the proposed enhancement is its simplicity, when compared with the existing model checking tools and techniques in the literature [13, 14]. Because the fault diagnoser is built from the software model itself and; since the modular approach is a must in the *Software Design Phase* of the V-model in EN 50128 (recommended as mandatory) there is no need for any tool to check the diagnosability of a simple software module (component) model [15]. The remainder of this paper is organized as follows. The V-model lifecycle and the modified V-model lifecycle are explained in Sections 2 and 3, respectively. DES-based fault diagnosis is introduced in Section 4, and conclusion section is given in Section 5.

2 V-model lifecycle

Paul Rook introduced the V-model lifecycle in 1986 as a guideline for software development processes [2]. The primary aim of the V-model is to improve both the efficiency of software development and the reliability of the produced software. The V-model offers a systematic roadmap from project initiation to product phase-out [2]. The V-model also defines the relationship between the development and test activities; it implements verification of each phase of the development process rather than testing at the end of the project. The V-model, as defined in IEC 61508-3, is shown in Figure 1.

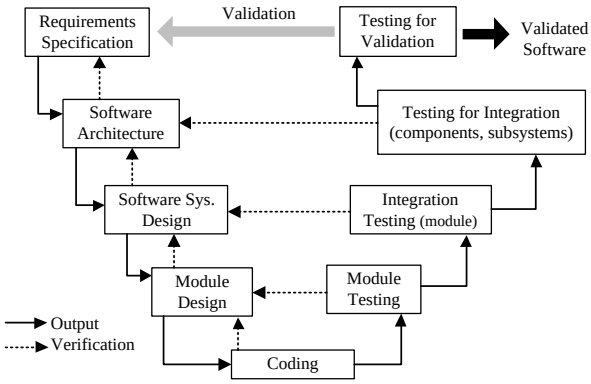

Figure 1: V-model software safety integrity and development lifecycle [16].

Before initializing a *software development process* according to the V-model, a *software planning phase* has to be realized, wherein a *software quality assurance plan*, *software verification* and *software validation plans*, and a *software maintenance plan* are fully defined. Later, the software requirements should be determined in cooperation with both the customer and the stakeholders. Using the selected software architectures (including modeling methods), software modules are developed by the designers. Each phase is verified immediately after completion. Note that, the left side of the V-model in Figure 1 represents the decomposition of the problem from the business world to the technical world [17]. After the coding phase, the right side of the V-model denotes the testing phase of the developed software.

The number of person may expand in the development process but this expansion shall be identified from the very beginning of the project. In [2], the change of the total number of software development teams are illustrated as given in Figure 2. Additionally, the cost of detection of faults in the different phases of V-model is given in Figure 3.

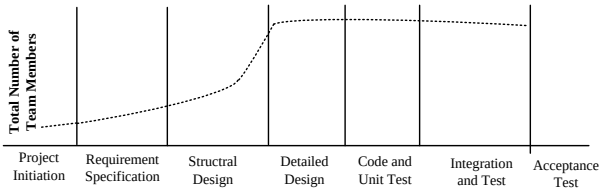

Figure 2: Software development teams [2].

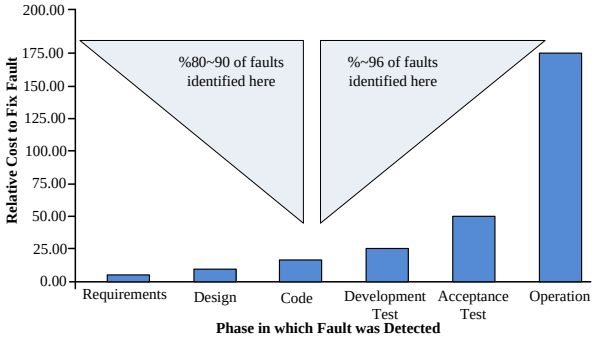

Figure 3: Software development teams [18].

The advantages and disadvantages of the V-model can be summarized as follows [4, 5]:

Advantages

1. Facilitates greater control due to the standardization of products in the process.
2. Cost estimation is relatively easy due to the repeatability of the process.
3. Each phase has specific products.
4. Greater likelihood of success because of the early development of test plans and documentation before coding.
5. Provides a simple roadmap for the software development process.

Disadvantages

1. Low flexibility, expensive, and difficult to change scope.
2. No early prototypes.
3. Addresses software development within a project rather than a whole organization.
4. Too simple to precisely reflect the software development process and may steer managers into a false sense of security.

3 Modified V-model lifecycle

As mentioned in [19], and [20], the required workforce and the cost of the development process of the software increases towards the end with respect to the initial phases of the development lifecycle. Therefore, the proposed modification is realized on the left side of the V-model.

In the usual software development process according to V-model, the fulfillment of the requirements are checked by realizing the module tests after coding. By checking the module diagnosability, one can decide if the module fully covers the software requirements related with faults or not (will be explained in section 4). This intermediate phase can be considered as time consuming and an extra workload. However, rather than turning back again from the module testing phase to the module design phase in the V-model, the proposed phase provides a final inspection of modules before proceeding to the coding and module testing phases. The proposed V-model is given in Figure 4.

The proposed modification in Figure 4 has three unique advantages:

a. *It checks whether the constructed model covers all software requirements related to faults*: If the developed software model (see Table A.2 and Table A.17 of [15]) is not diagnosable, then the software model does not contain all software requirements related with the faults.

b. *It decreases costs through early detection of modeling deficiencies before proceeding to coding and testing phases*: As can be seen from Figure 4, after proceeding to the coding phase, the designer can only go back to the module design phase at the end of the module tests. Many studies showed that, it is 5 times more expensive to fix a problem at the design stage than in the course of initial requirements, 10 times more expensive to fix it through the coding phase, 20 to 50 times more expensive to fix it at acceptance testing and, 100 to 200 times more expensive to fix that error in the course of actual operation [20-23].

c. *It enables designers to write simple and more readable code in decision of the faults*: This will be explained with a simple case study in the next section.

4 DES-based fault diagnosis

An *event* is defined as an encountered specific action, i.e., an unplanned incident that occurred naturally or due to numerous conditions that are encountered simultaneously [24]. Events are classified as observable or unobservable events in a DES.

A DES system is considered as diagnosable if it is possible to identify, within a finite delay, occurrences of precise unobservable events that are referred to as fault

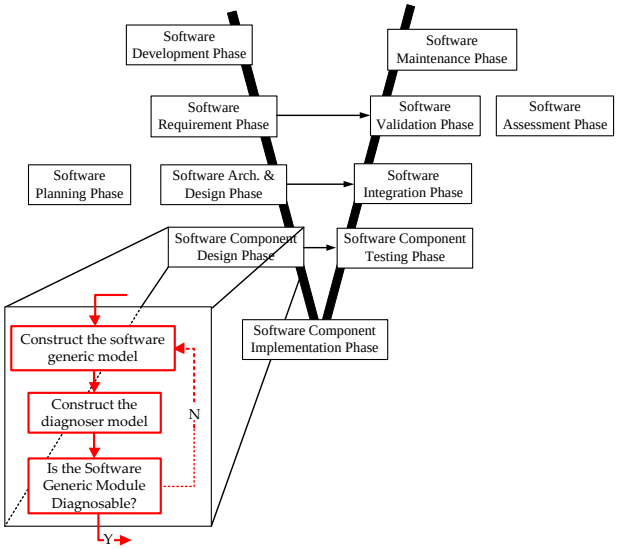

Figure 4: Enhanced V-model (Y-Yes, N-No).

events [25]. In other words, a system is diagnosable if the fault type is always identified within a uniformly bounded number of transition firings after the occurrence of the fault [26]. The diagnoser is obtained from the system model itself and carries out diagnostics to observe the system behavior. Diagnoser states involve fault information, and occurrences of faults are identified within a finite delay by examining these states [27].

Finite state machines and Petri nets are considered as DES-based modeling methods and, these methods are also highly recommended by functional safety standards (see [15]).

4.1 Basic petri net (*PN*) definitions

A Petri net [28] is defined as;

$$PN = (P, T, F, W, M_0) \tag{1}$$

where

- $P = \{p_1, p_2, \ldots, p_k\}$ is the finite set of places,
- $T = \{t_1, t_2, \ldots, t_z\}$ is the finite set of transitions,
- $F \subseteq (P \times T) \cup (T \times P)$ is the set of arcs,
- W: F $\rightarrow \{1, 2, 3, \ldots\}$ is the weight function,
- M_0: P $\rightarrow \{0, 1, 2, 3, \ldots\}$ is the initial marking,
- $P \cap T = \varnothing$ and $P \cup T \neq \varnothing$.

For a marking M, $M(p_i) = n$ represents the token number of the ith place where it is equal to n [28]. Representation of a marking $M : P \rightarrow \{1, 2, 3, ...\}$ can be realized by a k-element vector, where k denotes the total number of places.

Definition 1 [28]: If a *PN* has no self-loops, then it is considered as *pure* and when all arc weights of a *PN* are 1, then it is said to be *ordinary*.

Definition 2 [28]: $M_0[t_1 > M_1[t_2 > \ldots M_{k-1}[t_k > M_k$ means that, the marking M_k is reachable from the initial marking M_0 by the $t_1 t_2 \ldots t_k$ transitions sequence. Note

that, $R(M_0)$ denotes the set of all reachable markings from M_0.

Definition 3 [29]: A *PN* is said to be *free from deadlocks* if it is possible to find at least one enabled transition at every reachable marking.

The set of places, P is partitioned into a set of observable places and a set of unobservable places (P_o and P_{uo}). Likewise, the set of transitions, T is partitioned into a set of observable transitions and a set of unobservable transitions (T_o and T_{uo}). Thus, the partitioned sets for P and T can be expressed as

$$P = P_{uo} \cup P_o \text{ and } P_{uo} \cap P_o = \varnothing$$
$$T = T_{uo} \cup T_o \text{ and } T_{uo} \cap T_o = \varnothing \quad (2)$$

In addition, a subset T_F of T_{uo} represents a faulty transitions set. It is assumed that there are m different fault types. Here, $\Delta_F = \{F_1, F_2, \ldots, F_m\}$ is the set of fault types. T_F is expressed as $T_F = T_{F_1} \cup T_{F_2} \cup \cdots \cup T_{F_m}$, where $T_{F_i} \cap T_{F_j} = \varnothing$ (if $i \neq j$).

$\Delta = \{N\} \bigcup 2^{\Delta_F}$ is used to define the label set, where N is used to represent the label "normal," which specifies that all fired transitions are *not faulty*, and 2^{Δ_F} represents the power set of Δ_F. In the remainder of this paper, unobservable transitions and places are represented as shown in Figure 5.

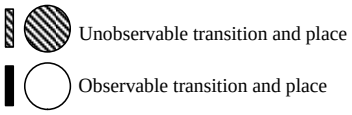

Figure 5: Representation of places and transitions.

4.2 Diagnosis of faults by using *PN* models

Since the system model contains unobservable places, it is not always possible to distinguish some markings. Thus, if $M_1(p_i) = M_2(p_i)$ for any $p_i \in P_o$, then it is denoted as $M_1 \equiv M_2$. That is to say, M_1 and M_2 markings have the same observations. As done in [30], the definition of the quotient set $\hat{R}(M_0)$ according to the equivalence relation $(\equiv)$ is useful; $\hat{R}(M_0) := R(M_0)/\equiv := \{\hat{M}_0, \ldots, \hat{M}_n, \ldots\}$ where $M_0 \in \hat{M}_0$. An observable marking or the observation of a marking is represented by each member of $\hat{R}(M_0)$. We assume the following two statements are true for simplicity.

Assumption [25, 26]: Only *deadlock free PN*s are considered and there does not exist an order of unobservable transitions whose firing produces a cycle of markings that have the same observation.

A diagnoser is given for a *PN* [26, 27] by

$$G_d = (Q_d, \Sigma_o, \delta_d, q_0). \quad (3)$$

The diagnoser given by (3) is an automaton where the set of states are represented by $Q_d \subseteq Q$, the set of events are represented by $\Sigma_o = \hat{R}(M_0) \cup T_o$, the notation $\delta_d : Q_d \times \Sigma_o \to Q_d$ represents the partial state transition function, and the initial state is denoted by $q_0 = \{(M_0, N)\}$. A diagnoser state q_d is given as $q_d = \{(M_1, l_1), (M_2, l_2), \ldots, (M_n, l_n)\}$, which involves pairs of a marking $M_i \in R(M_0)$ and a label $l_i \in \Delta$. The state set $Q_d \subseteq Q$ represents the reachable states from the initial state q_0 by using δ_d. Each observed event $\sigma_o \in \Sigma_o$ represents an observation of a marking in $\hat{R}(M_0)$ or an observable transition in T_o. The state transition function δ_d is defined with the use of the label propagation function and the range function.

The label propagation function associates a label (*faulty* or *normal*) over a sequence of transitions. If the sequence of transitions does not contain any faulty transition, the resulting marking is labeled as normal (N). Detailed explanation of the label propagation function, the range function and the state transition function can be seen from [25-27, 30, 31].

4.3 Obtaining diagnosability

A *PN* is diagnosable if, and only if, the states of the diagnoser given by (3) shall be F_m-certain or does not involve any F_m-indeterminate cycle for any fault type F_m. Due to page restriction, the reader is referred [25-27] for detailed explanation of DES-based fault diagnosis and the proof of this theorem.

4.4 Railway point example

Trains can move from one track to another by the help of railway *points* (rail switches or point machines) placed at necessary locations. Points have two position indications, i.e., Normal (*Nr*) and Reverse (*Rev*). At any railway point, three main faults may occur. These faults are identified in the V-model software requirements specification phase as follows:

- F_1: Point may not reach the desired position in a predefined time (e.g., 5 sec) while moving from *Nr* to *Rev*.
- F_2: Point may not reach the desired position in a predefined time while moving from *Rev* to *Nr*.
- F_3: Both position indications may be received simultaneously.

Examples of diagnosable and not diagnosable *PN* models of a railway point are given in Figure 6 and Figure 7, respectively. The meanings of the transitions and places of the models in Figure 6 and Figure 7 are given in Table 1 and Table 2, respectively. Note that the striped places and transitions represent unobservable places (P_{uo}) and transitions (T_{uo}), whereas the other places (P_o) and transitions (T_o) are observable. M_0 represents the initial marking of the *PN*. The underlined numbers in the diagnoser are used to represent the marking of an unobservable place.

Representation of the *PN* model in Figure 6 is as follows:

$$P_o = \{P_{PM\_1}, P_{PM\_2}, P_{PM\_5}, P_{PM\_6}, P_{PM\_8}, P_{PM\_10}, P_{PM\_12}\},$$
$$P_{uo} = \{P_{PM\_3}, P_{PM\_4}, P_{PM\_7}, P_{PM\_9}, P_{PM\_11}\},$$
$$T_o = \{t_{PM\_1}, t_{PM\_2}, t_{PM\_3}, t_{PM\_4}, t_{PM\_5}, t_{PM\_6}, t_{PM\_7}, t_{PM\_8}, t_{PM\_9}, t_{PM\_10}, t_{PM\_11}, t_{PM\_12}, t_{PM\_13}, t_{PM\_14}\}, \quad T_{uo} = \{t_{PM\_f1}, t_{PM\_f2}, t_{PM\_f3}\}, \quad (4)$$
$$M_0 = (M_0(P_{PM\_1}), M_0(P_{PM\_2}), M_0(P_{PM\_3}), M_0(P_{PM\_4}), M_0(P_{PM\_5}), M_0(P_{PM\_6}), M_0(P_{PM\_7}), \ldots$$
$$\ldots, M_0(P_{PM\_8}), M_0(P_{PM\_9}), M_0(P_{PM\_10}), M_0(P_{PM\_11}), M_0(P_{PM\_12})) = (0,0,\underline{0},\underline{0},1,0,\underline{0},0,\underline{0},0,\underline{1},0).$$

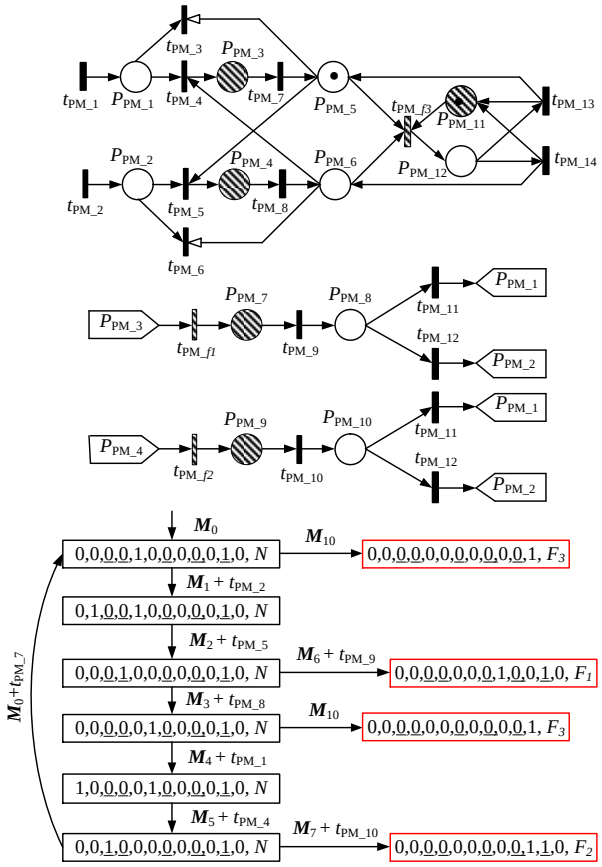

Figure 6: PN model of a railway point and its diagnoser (*diagnosable*).

Three different fault types given in Figure 6 are $\Delta_F = \{F_1, F_2, F_3\}$, where $T_{F_1} = \{t_{PM\_f1}\}$, $T_{F_2} = \{t_{PM\_f2}\}$, and $T_{F_3} = \{t_{PM\_f3}\}$. The rectangles are used to diminish the complexity of the *PN* model. Each rectangle represents the label of the related place.

The diagnoser illustrated in Figure 6 is built from the *PN* model of railway point. A rectangle is used to denote each state and each state contains a pair of place markings and an attached label, normal (*N*) or fault. In other words, in parts of the diagnoser, a marking immediately after an observed event is detected precisely.

In accordance with the definition of the diagnoser in (3), a label which represents an observable transition or the observation of a marking is attached to all diagnoser state transitions.

In this study, with a slight abuse of notation, labels containing the observation of a marking or a pair of the observation of a marking and an observable transition are attached to all state transitions of the diagnoser.

Place	Definition	Transition	Definition
P_{PM_1}	*Nr* position requested	t_{PM_1}	Point *Nr* position request
P_{PM_2}	*Rev* position requested	t_{PM_2}	Point *Rev* position request
P_{PM_3}	Point is moving to *Nr*	t_{PM_3} (t_{PM_6})	Request ignored
P_{PM_4}	Point is moving to *Rev*	t_{PM_4}	Point left *Rev*
P_{PM_5}	Point is in *Nr*	t_{PM_5}	Point left *Nr*
P_{PM_6}	Point is in *Rev*	t_{PM_7} (t_{PM_8})	Point reached to *Nr* (*Rev*)
P_{PM_7}	Fault type F_1 has occurred	t_{PM_9} (t_{PM_10})	Filter time has expired
P_{PM_8}	Point is faulty (F_1)	t_{PM_11} (t_{PM_12})	*Nr* (*Rev*) position request
P_{PM_9}	Fault type F_2 has occurred	t_{PM_13} (t_{PM_14})	Point moved to *Nr* (*Rev*) and the fault acknowledged
P_{PM_10}	Point is faulty (F_2)	t_{PM_f1}	Point indication fault
P_{PM_11}	Unobservable fault restriction	t_{PM_f2}	Point indication fault
P_{PM_12}	Point is faulty (F_3)	t_{PM_f3}	Point position fault

Table 1: Definition of transitions and places in the models given in Figure 6.

For example, at $\hat{M}_0$ in Figure 6, the event label $\hat{M}_{10}$ represents that the observable marking $\hat{M}_{10}$ is observed by firing the unobservable transition t_{PM_f3}. Similarly, the diagnoser state changes by firing the unobservable transition t_{PM_f3}. Similarly, the diagnoser state changes from $\{((0,0,\underline{0},\underline{0},1,0,\underline{0},0,\underline{0},0,\underline{1},0,),N)\}$, to $\{((0,1,\underline{0},\underline{0},1,0,\underline{0},0,\underline{0},0,\underline{1},0),N)\}$, as a function of firing the observable transition t_{PM_2} with the observation $\hat{M}_1$ of the resulting marking. According to the definition given in Section 4.3, since all states are F_m-certain and there is no F_m-indeterminate cycle in the diagnoser, the *PN* model is diagnosable.

Representation of the *PN* model in Figure 7 is as follows:

$$
\begin{aligned}
&P_o = \{P_{PM\_1}, P_{PM\_2}, P_{PM\_3}, P_{PM\_4}, P_{PM\_5}, P_{PM\_6}\}, \quad P_{uo} = \{P_{PM\_7}\},\\
&T_o = \{t_{PM\_1}, t_{PM\_2}, t_{PM\_3}, t_{PM\_4}, t_{PM\_5}, t_{PM\_6}, t_{PM\_7}, t_{PM\_8}\}, \quad T_{uo} = \{t_{PM\_f1}, t_{PM\_f2}, t_{PM\_f3}\},\\
&M_0 = (M_0(P_{PM\_1}), M_0(P_{PM\_2}), M_0(P_{PM\_3}), M_0(P_{PM\_4}), M_0(P_{PM\_5}), M_0(P_{PM\_6}), M_0(P_{PM\_7}))\\
&\quad = (0,0,1,0,0,0,\underline{0}).
\end{aligned} \tag{5}
$$

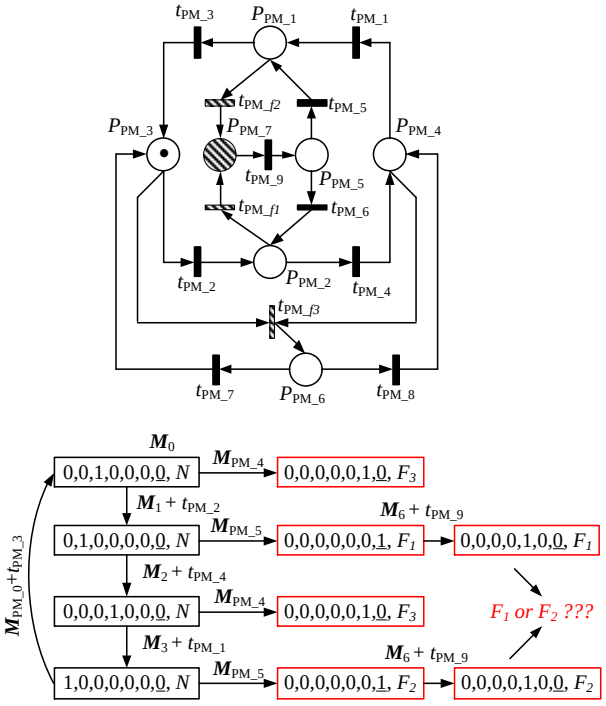

Figure 7: PN model of a railway point and its diagnoser (*not diagnosable*).

The diagnoser given in Figure 7 is not diagnosable because it is not possible to distinguish the fault type after observing the marking $\hat{M}_5$. In this case, the *PN* model will identify only one of the faults (F_1 or F_2) while the obtained code from this *PN* model is running. Therefore, the designers should revise the *PN* model before proceeding to the coding phase; otherwise, this deficiency will result in an unsuccessful test case in the module testing phase.

4.5 Railway signal example

An example *PN* model of a Two-Aspect Signal (TAS) and its diagnoser is given in Figure 8. TAS is generally used in railway depot areas and has two signal color indications (red means stop and green means proceed). The meanings of the transitions and places of the model in Figure 8 are given in Table 3. It is assumed that two different faults may occur in TAS which are F_1: Both signal aspects are lit at the same time; and F_2: No signals are lit.

Place	Definition	Transition	Definition
P_{PM_1}	Point is moving to *Nr*	t_{PM_1} (t_{PM_2})	Movement request is received for *Nr* (*Rev*) position
P_{PM_2}	Point is moving to *Rev*	t_{PM_3} (t_{PM_4})	Point reached to *Nr* (*Rev*)
P_{PM_3}	Point position is *Nr*	t_{PM_5} (t_{PM_6})	Point request to *Nr* (*Rev*)
P_{PM_4}	Point position is *Rev*	t_{PM_7} (t_{PM_8})	Point moved to *Nr* (*Rev*) and the fault acknowledged
P_{PM_5}	Fault type F_1 or F_2 has occurred	t_{PM_9}	Filter time has expired
P_{PM_6}	Point is faulty (F_3)	t_{PM_f1} (t_{PM_f2})	Point indication fault
P_{PM_7}	Point is moving from one position to another	t_{PM_f3}	Point position fault

Table 2: Definition of transitions and places in the models given in Figure 7.

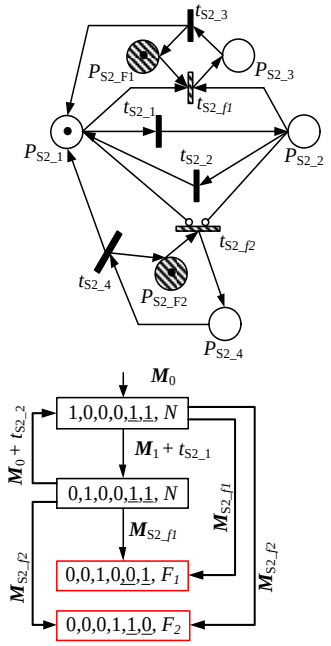

Figure 8: *PN* model of TAS and its diagnoser.

Place	Definition	Transition	Definition
P_{S2_1}	Signal is red	t_{S2_1}	Turn signal to green

P_{S2_2}	Signal is green	t_{S2_2}	Turn signal to red
P_{S2_3}	Fault type F_1 has occurred	t_{S2_3}	Signal turned to red and the fault acknowledged
P_{S2_4}	Fault type F_2 has occurred	t_{S2_4}	Signal turned to red and the fault acknowledged
P_{S2_F1}	Unobservable fault restriction	t_{S2_f1}	Point aspect fault
P_{S2_F2}	Unobservable fault restriction	t_{S2_f2}	Point indication fault

Table 3: Definition of transitions and places in the models given in Figure 8.

To compare the simplicity in decision of the faults with and without a diagnoser, an example Programmable Logic Controller (PLC) code snippet of TAS model is shown in Figure 9 and Figure 10, respectively.

As can be seen from Figure 9 and Figure 10, decision of fault occurrence with a diagnoser is simpler than without a diagnoser. For the PLC code given in Figure 9, the diagnoser compares the actual states of the *PN* model with predefined faulty states. When the faulty state of the diagnoser is fully matched with the marking of the actual *PN* states, the diagnoser sets the corresponding fault label.

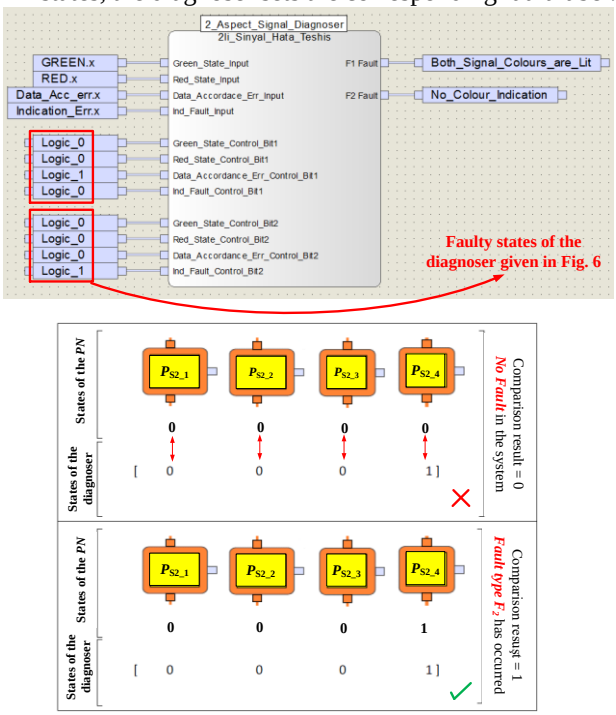

Figure 9: Diagnoser block of TAS and decision of fault occurrence.

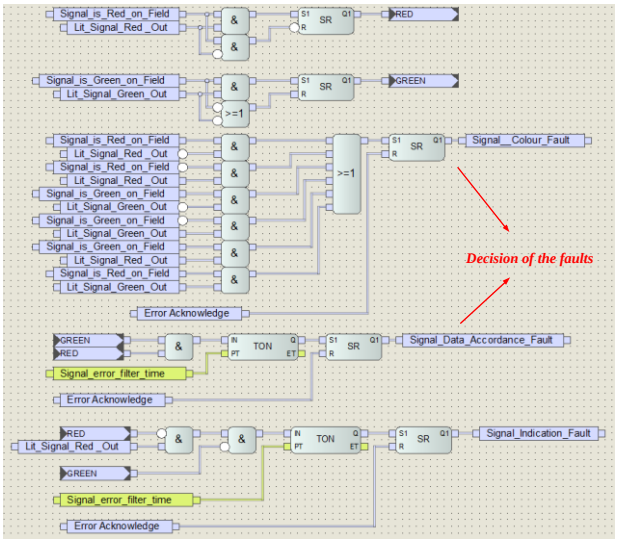

Figure 10: Decision of fault for TAS without diagnoser.

Moreover, since the V-model is modified by adding an additional step, we also defined a new task for the organizational structure of the software development team. The Diagnoser designer (DDes) is added to the preferred organizational structure of the enhanced V-model as given in Figure 11 (PM: Project Manager, RQM: Requirement Manager, Des: Designer, IMP: Implementer, VER: Verifier, VAL: Validator, DDes: Diagnoser Designer, ASR: Assessor). The original organizational structure can be seen from EN 50128 [15].

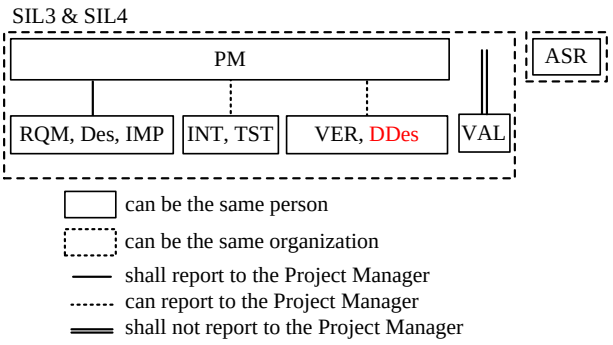

Figure 11: Recommended organizational structure for the enhanced V-model for SIL3&SIL4 software.

5 Conclusion

Faults in a safety-critical system may cause severe harm to humans. Therefore, the development steps of software for such safety-critical systems must be executed very carefully. Designers, developers, and engineers must consider the recommendations of both the international safety standards and the national rules to satisfy the required safety level and fulfill requirements.

Although enhancing the V-model with DES-based fault diagnosis is time consuming, however, the advantages of this intermediate step are threefold: (1) it checks whether the developed model fulfills all software requirements related to the faults; (2) decision of faults with a diagnoser is simpler than without a diagnoser; and (3) an early check of the models is possible before

proceeding to the coding and testing phase because the V-model leads developers from the module testing phase to the module design phase rather than the coding phase.

On the other hand, when costs and work hours are considered, adding such an intermediate step to the V-model can result in considerable benefits to both project management and product development departments.

Acknowledgement

The authors are thankful to Enago (www.enago.com) for the review of the English language of the paper.

References

[1] IEC61508 (2010). *Functional safety of electrical/electronic/programmable electronic safety-related systems, Parts 1–7*. International Electrotechnical Commission.

[2] Rook P (1986). Controlling Software Projects. *Software Engineering Journal*, 1, pp. 7-16. https://doi.org/10.1049/sej.1986.0003

[3] IEC 61508-4 (2010). *Functional safety of electrical/electronic/programmable electronic safety-related systems, Part 4: Definitions and Abbreviations*. International Electrotechnical Commission.

[4] Munassar NM, Govardhan A (2010). A Comparison Between Five Models of Software Engineering. *International Journal of Computer Science Issues*, 7, pp. 94-101.

[5] Krishna ST, Sreekanth S, Perumal K, Kumar Reddy KR (2012). Explore 10 Different Types of Software Development Process Models. *International Journal of Computer Science and Information Technologies*, 3:4580-4584.

[6] Royce WW (1970). Managing the Development of Large Software Systems: Concepts ad Techniques. *Proceedings Wescon*, pp. 1-9.

[7] Boehm BW (1988). A Spiral Model of Software Development and Enhancement. *Computer*, 21, pp. 61-72. https://doi.org/10.1109/2.59

[8] Lehman MM (1980). Programs, Life Cycles, and Laws of Software Evolution. *Proceedings of the IEEE*, 68, pp. 1060-1076. https://doi.org/10.1109/PROC.1980.11805

[9] Rahman RA, Pulm U, Stetter R (2007). Systematic Mechatronic Design of a Piezo-Electric Brake. *16th International Conference on Engineering Design*, 28-31 July, Paris, France, pp. 1-12.

[10] Märtin L, Schatalov M, Hagner M, Goltz U, Maibaum O (2013). A Methodology for Model-Based Development and Automated Verification of Software for Aerospace Systems. *IEEE Aerospace Conference*, 2-9 March, Big Sky, MT, USA, pp. 1-19. https://doi.org/10.1109/AERO.2013.6496950

[11] Scippacercola F, Pietrantuono R, Russo R, Zentai A (2015). Model-Driven Engineering of a Railway Interlocking System. *3rd Int Conf on Model-Driven Eng and Soft Development*, 2-9 September, Angers, France, pp pp. 509-519. https://doi.org/10.1007/978-3-319-27869-8_22

[12] SSG-39 (2016). *Design of Instrumentation and Control Systems for Nuclear Power Plants*. IAEA Safety Standards Series.

[13] Kwiatkowska M, Norman G, Parker D (2002). PRISM: Probabilistic Symbolic Model Checker. Field T, Harrison PG, Bradley J, Harder U (ed) *Computer Performance Evaluation: Modeling Techniques and Tools, Lecture Notes in Computer Science*, Springer, Berlin, Heidelberg, pp. 200-204. https://doi.org/10.1007/3-540-46029-2_13

[14] Holzmann GJ (2003). *Spin model checker, the: primer and reference manual*. Addison-Wesley.

[15] BS EN 50128 (2011). *Railway Applications-Communication, Signalling and processing systems: Software for railway control and protection systems*. International Electrotechnical Commission.

[16] IEC 61508-3 (2010). *Functional safety of electrical/electronic/programmable electronic safety-related systems, Part 3: Software Requirements*. International Electrotechnical Commission.

[17] Ratcliffe A (2011). SAS Software Development with the V-Model. *3SAS Global Forum, Coder's Corner*, 4-7 April, Las Vegas, Nevada, USA, pp. 1-9.

[18] Brat GP (2017). Reducing V&V Cost of Flight Critical Systems: Myth or Reality? *AIAA Information Systems, AIAA SciTech Forum*, American Institute of Aeronautics and Astronautics, 9-13 January, Grapevine, Texas, USA, pp. 1-10.

[19] Boehm BW (1984). Verifying and Validating Software Requirements and Design Specifications. *IEEE Software*, 1, pp. 75-88. https://doi.org/10.1109/MS.1984.233702

[20] Boehm BW (1984). Software Engineering Economics. *IEEE Transactions on Software Engineering*, SE-10, pp. 4-21. https://doi.org/10.1109/TSE.1984.5010193

[21] Boehm BW (1987). Industrial Software Metrics: A Top Ten List. *IEEE Software*, 4, pp. 264-271.

[22] Haskins B, Stecklein J, Dick B, Moroney G, Lovell R, Dabney J (2004). Error Cost Escalation Through the Project Life Cycle. *14th Annual Int Symp, Int Council on Systems Engineering*, 19-24 June, Toulouse, France, pp. 1723-1737. https://doi.org/10.1002/j.2334-5837.2004.tb00608.x

[23] Schneider GM, Martin J, Tsai WT (1992). An Experimental Study of Fault Detection in User Requirements Documents. *IACM Transactions on Software Engineering and Methodology*, 1, pp. 188-204. https://doi.org/10.1145/128894.128897

[24] Cassandras CG, Lafortune S (2008). *Introduction to Discrete Event Systems*. Springer, New York. https://doi.org/10.1007/978-0-387-68612-7

[25] Sampath M, Sengupta R, Lafortune S, Sinnamohideen K, Teneketzis D (1995). Diagnosability of discrete-event systems. *IEEE Trans on Automatic Control*, 40, pp. 1555-1575.

https://doi.org/10.1109/9.412626

[26] Ushio T, Onishi I, Okuda K (1998). Fault detection based on Petri net models with faulty behaviours. *International Conference on Systems, Man, and Cybernetics*, 11-14 October, San Diego, CA, USA, pp. 113-118.

[27] Sampath M, Sengupta R, Lafortune S, Sinnamohideen K, Teneketzis D (1996). Failure diagnosis using discrete-event models. *IEEE Transactions on Control Systems Technology*, 4, pp. 105-124. https://doi.org/10.1109/87.486338

[28] Murata T (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77, pp. 541-580. https://doi.org/10.1109/5.24143

[29] Li ZW, Zhou MC, Wu NQ (2008). A survey and comparison of Petri net-based deadlock prevention policies for flexible manufacturing systems. *IEEE Trans on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 38, pp. 173–188. https://doi.org/10.1109/TSMCC.2007.913920

[30] Chung SL (2005). Diagnosing PN-based models with partial observable transitions. *International Journal of Computer Integrated Manufacturing*, 18, pp. 158-169. https://doi.org/10.1080/0951192052000288206

[31] Durmuş MS, Takai S, Söylemez MT (2014). Fault Diagnosis in Fixed-Block Railway Signaling Systems: A Discrete Event Systems Approach. *IEEJ Transactions on Electrical and Electronic Engineering*, 9, pp. 523-531. https://doi.org/10.1002/tee.22001