SCHOOL OF ENGINEERING

THE UNIVERSITY
OF BIRMINGHAM

SIMULATION COMBINED MODEL-BASED TESTING METHOD FOR TRAIN CONTROL SYSTEMS

By

YUEMIAO WANG

A thesis submitted to the University of Birmingham

for the degree of

DOCTOR OF PHILOSOPHY

12th March 2018

Abstract

A Train Control System (TCS) is utilised to guard the operational safety of the trains in railway systems. With rapid developments in modern railway systems, more and more modern TCSs have been developed to protect system operation. Consequently, guaranteeing that the functions of a TCS satisfy the designed specification requirements is essential to affirm that a developed TCS can be adopted. Functional testing is applied to test the System Under Test (SUT) in order to verify consistency between the SUT and specification requirements. Traditional functional testing in TCSs is mainly based on manually designed test cases, which are derived from experienced experts who are familiar with system functional design and testing. For newly built or updated TCSs, the test case generation process can take a long time. Manually-written test cases may miss some scenarios that should have been tested, even when prepared by an experienced test designer. Model-Based Testing (MBT) methods have been introduced into TCS functional testing to improve the efficiency and coverage of TCS testing. However, existing MBT methods cannot independently test complex SUTs because the model complexity generated by the SUT can exceed the computational limit of the computer due to state explosion.

To overcome the difficulties of applying MBT methods to test TCSs, the author introduces simulation combined MBT which combines an MBT method with simulation. To explain the MBT method introduced, related background knowledge is reviewed. Due to the limitations of the current functional testing and MBT methods, the author describes the research problem, and proposes methodology and development of the simulation combined MBT method, and

the validation and verification of the testing platform.

To prove the feasibility and effectiveness of the proposed MBT method and developed MBT platform, two case studies were undertaken. The test results indicate that the SUT Vehicle On-Board Controller (VOBC) complies with the specification requirements so that it passed the test. The two case studies prove that the developed MBT platform can be utilised to implement the functional testing of TCSs.

To prove that the MBT platform is effective in detecting errors in the SUT, validation and verification was undertaken, which included validation of the specification requirements and verification of the MBT platform. The verification results indicate that the MBT platform can cover more possible traces and variable values at the same search depth. The author also explores the possibilities of improving the coverage performance of the platform by improving its reachset coverage in key states. Various impact factors have been discovered to be effective in making the platform cover more possibilities in the same testing time.

Acknowledgements

I would like to give my sincere appreciation to my supervisors, Dr Lei Chen and Prof. Clive Roberts, for their consistent guidance, support and encouragement during my PhD study. With their help, I have moved firmly and in the correct direction toward my goal. The kind advice and patience they have given me has increased my confidence to overcome difficulties encountered in my PhD research. The knowledge they have given me has brought me benefits in my research and future career.

I would like to extend my gratitude to the Dr Jidong Lv who has selflessly shared his experience and knowledge with me, helping me successfully find the direction for my research. I am also grateful to Dr David Kirkwood who has shared his professional experiences and skills in Java programming and railway simulation with me. I would like to thank Miss Katherine Slater for her great help for proof reading my academic writing. Many thanks go to all the members of the Birmingham Centre for Railway Research and Education for their help and support.

I am also grateful to my wife, parents and father-in-law for their great love and understanding. Your love is the greatest motivation in my life.

Finally, I would like to express sincere appreciation of my much-missed mother-in-law, an honest and selfless woman who lives in our hearts forever.

Table of Contents

List of Figures

List of tables

Abbreviations

ATP	Automatic Train Protection
ATS	Automatic Train Supervision
BTM	Balise Transmission Module
CBI	Computer-Based Interlocking
CBTC	Communication-Based Train Control
CCS	Calculus of Communication System
CRRC	China Railway Rolling Stock Corporation
CSP	Communicating Sequential Processes
CTC	Centralised Traffic Control
DCS	Data Communication System
DMI	Driver Machine Interface
DR	Data Recorder
EB	Emergency Brake
ETCS	European Train Control System
EVC	European Vital Computer
FFFIS	Form Fit Functional Interface Specification
FSM	Finite State Machine
GSM-R	Global System for Mobile Communications – Railway
HIL	Hardware-in-the-Loop
HOL	Higher-Order Logic

IATP	Intermissive Automatic Train Protection
ioco	Input–output conformance relation
IOTS	I/O Transition System
IUT	Implementation Under Test
JML	Java Modelling Language
JRU	Juridical Recording Unit
KVC	Kernel Vital Computer
LEU	Lineside Electronic Unit
LTS	Labelled Transition Systems
MA	Movement Authority
MBT	Model-Based Testing
MSC	Message-Sequence Charts
OBRU	On-Board Radio Unit
OBU	On-Board Unit
OCL	Object Constraint Language
ODO	Odometer
PN	Petri Net
RBC	Radio Block Centre
RTM	Radio Transmission Module
SCTIOTS	Simulation Combined Model-Based Testing
SIL	Safety Integration Level

sioco	Symbolic input–output conformance relation
SUT	System Under Test
TA	Timed Automata
TC	Track Circuit
TCC	Train Control Centre
TCR	Track Circuit Reader
TCS	Train Control System
tioco	Timed input–output conformance relation
TIA	Transponder Interrogator Antennae
TIMS	Train Information Management System
TIOTS	Timed I/O Transition System
TIU	Train Interface Unit
TOD	Train Operator Display
UML	Unified Modelling Language
UNISIG	Union of Signalling Industry
VDM	Vienna Development Method
VOBC	Vehicle On-Board Controller
WLAN	Wireless Local Area Network
ZC	Zone Controller

1 Introduction

1.1 Background

The Train Control System (TCS) is a wide-ranging concept with numerous subsystems for different control objectives, such as guaranteeing system safety, improving efficiency and capacity, and optimising energy consumption [1-3]. Among all these subsystems, the TCS is one of the most essential because it is the key element to guaranteeing a system's operational safety and protecting the system from potential dangerous situations such as collision or derailment [4, 5]. Since the railway is utilised to carry a large quantity of passengers or cargo, any dangerous situation can lead to disastrous consequences and huge economic losses. With the same purpose for different uses, there is much variation in the signalling systems adopted in different countries, such as the European Train Control System (ETCS) which is a unified standard and widely adopted in the railway systems of many European countries [6, 7], and the Communication-Based Train Control (CBTC) system which has been widely adopted in many countries.

Since metro lines are usually less complicated than mainline railways in terms of track layout, rolling stock and timetables, moving blocks has been realised in CBTC systems to improve the capacity of metro operations [8, 9]. Different from the ETCS standard which have unified standards determined by authoritative organisations, CBTC system composition varies for different manufacturers, including different system components, structure and performance. Despite ETCS and CBTC being different from each other in many different aspects, they still share a lot in common. For example, the fundamental system structure of each contains

trackside equipment, train-borne equipment, and communication systems, which are used to guarantee the safety of train movements with determined train movement authority (MA) [10, 11]. Based on data transmission between lineside equipment and train-borne equipment via communication channels [12], train movement safety is guaranteed by cooperation of these essential elements, though the cooperation modes may be different in different standards of TCS [13]. Therefore, it is possible to apply a unified method to test different types of signalling system.

As one of the most essential protectors of railway systems operation, TCSs are required to contain no safety-relevant errors that could lead the system operation into dangerous situations [14]. As a result, TCSs consist of a series of Safety Integration Level (SIL) 3 and SIL 4 [15] subsystems and components, which makes a TCS a typical safety-critical system which must be fail-safe [16]. Therefore, functional testing plays an important role in verifying that all safety-related functions in TCSs are correctly designed and precisely realised. To achieve this goal, test cases are written to check against the system specification requirements, aiming to determine whether inconsistencies exist between the system specification and the System Under Test (SUT) [17]. To implement testing, test cases need to be drafted by experts in the testing field, who also need to be experienced in signalling system design. Based on this understanding of a certain signalling system, test cases are written to include a series of different scenarios in which failure or dangerous situations could happen during system operation. This procedure requires the test case drafter to completely master the whole system operation process so that he or she knows every function which needs to be tested. After test

case drafting is finished, the test cases must be translated into a set of test sequences which describe what actually happens in the testing procedure. Finally, the last step is to execute the test sequences generated to determine whether the SUT behaviour complies with the system specification requirements. The test process can be time-consuming because most steps in the process are performed manually, as shown in Fig 1:

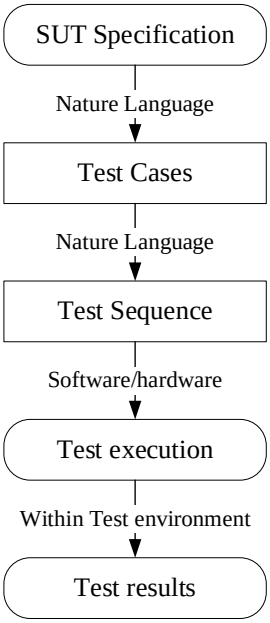


Fig 1 General steps of manual testing

According to Fig 1, test cases are written in natural language, according to the specification requirements of the SUT and the expertise of the tester. The test cases are then manually translated into test sequences which are also in natural language, describing the testing steps which happen in the testing process. To execute the testing sequence and eventually obtain the testing results, the testing sequences have to be transformed into a format which is recognisable by the SUT software or hardware; this can be realised by specific interfaces which can realise the translation between different types of data. Finally, the tests are implemented in a valid testing environment, and the testing results are obtained. Due to the

SUT specification requirements, the test case and the test sequence are all in natural language which is understandable for humans but is difficult for computers to process; testing efficiency relies significantly on the person who designs and implements the testing. Therefore, it is extremely difficult to improve testing performance, due to the human factor. Even worse, due to the complexity of TCSs, manually drafted test cases can miss essential testing steps and cause error omissions, even with experienced testers. If the system specification is modified in the system development stage, the test case must be accordingly modified to comply with the specification requirements, which means the test sequences and test executions must be modified as well. Without the assistance of computers, the modification process can take an extremely long time so that the extendibility of the testing can be reduced. Overall, manually oriented testing has become less appropriate for modern TCSs due to the growing demands for quicker product delivery with high quality [18].

Faced with the conflicts between the manual testing method and the requirements for functional testing of TCSs, automatic testing methods have been taken into the field from software testing [19]. Unlike manual testing, automatic testing methods can automatically generate test cases based on the formalised specification requirements of the SUT so that testing quality and duration can be significantly reduced with the assistance of computers [20, 21]. As cutting-edge technology in the testing field, automatic testing has a wide range of realisation methods for different testing objectives, including stress testing [22, 23], usability testing [24], performance testing [25], functional testing, etc.

To verify the functional correctness of the system, which is one of the key tasks in the testing

of TCSs, Model-Based Testing (MBT) is one of the most common automatic testing methods for functional testing [26, 27]. To implement MBT, the specification requirements need to be formally described by a specification model which is readable by a computer. The specification model is then analysed by the computer with integrated algorithms, and corresponding test cases are generated based on the properties that need to be verified. Compared with the manual testing method, the MBT method has several advantages which mean it can replace the current manual testing method utilised in TCS testing. Firstly, MBT test generation can be achieved as soon as the specification requirements are formally presented, which means the whole testing process period can be significantly shortened. Compared with test cases which are written in natural language, the formal models that describe an SUT according to its specification requirements are more precise because formal language is more logical and mathematical and has less ambiguity. This is extremely important for the testing of safety-critical systems such as TCSs because even a slight misunderstanding of the specification requirements can result in an incorrect testing verdict, leading to serious consequences [28]. That is why more and more manual testing adopts formal language to describe the specification requirements of the SUT, even though the test cases are still executed manually. Secondly, with MBT test generation algorithms based on formalised specification requirements, coverage of the generated test cases can be conveniently calculated and improved by the algorithm so that testing efficiency and coverage can be dramatically improved [29]. The research results of Utting and Legeard [30] show the significance of automating the process of functional testing.

Activity	Total testing time				
	Manual	Replay	Script	Keyword	MBT
Test design	50	50	50	50	0
Modelling	-	-	-	-	30
Initial configuration	-	-	50	15	30
Initial test execution	30	30	2	2	2
Total testing time by version 1	80	80	102	67	62
Total testing time by version 2	118	103	122	82	76
Total testing time by version 3	160	128	143	97	90
Total testing time by version 4	206	156	166	113	104
Total testing time by version 5	257	187	191	129	118
Total testing time by version 6	313	221	219	146	132
Total testing time by version 7	374	258	249	164	146
Total testing time by version 8	441	299	282	182	160
Total testing time by version 9	515	344	318	201	174
Total testing time by version 10	596	393	358	221	188

Table 1 Testing time for each version of testing for each testing method [30]

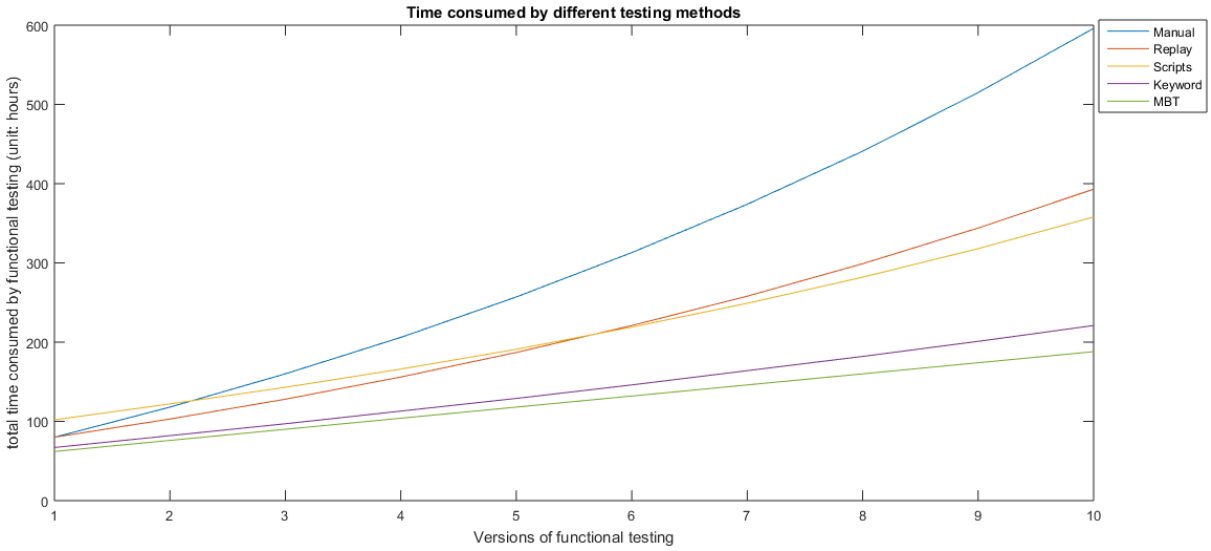


Fig 2 Efficiency comparison of different testing methods [30]

As revealed by Table 1 and Fig 2, the total test duration increases along with an increasing

number of versions, where more and more functions are included in the testing so that more and more test cases need to be generated and executed. Obviously, for the initial versions, the duration of the five testing methods do not differ a lot compared with the following versions. However, the growth rates of manual testing, replay testing [31] and script-based testing [32, 33] are much greater than those of keyword-driven testing [34] and MBT. As a result, when the testing versions are updated, which means that more and more functions are tested, keyword-driven testing and MBT can save a large amount of testing time compared to other testing methods. Since MBT is the only testing method that can automate the test design process, its advantages against keyword-driven testing appear when more and more functions need to be tested. For complex systems such as TCSs, the number of functions under test can be high so that MBT can play its strengths when testing such kinds of system. Furthermore, in manual testing, a single test sequence is assigned to one test case to ensure that the corresponding function in the SUT is covered by the testing. However, there can be a lot of different testing sequences contained in the same test case, which means that in some instances, one cannot stand for all of them. In manual testing, there are too many remaining valid test sequences to be fully covered by human design. Since formal models can be understood by computers, a computer can find out all the valid test cases from the specification requirements, and all the test sequences based on the test cases generated. As a result, coverage of the testing can be significantly improved with MBT test generation algorithms. Furthermore, the test cases generated can be easily transformed into test sequences, which can be used to realise automatic test execution with a specified interface

connecting the real SUT and the test tool. Consequently, the test execution efficiency can be greatly improved without any risk of affecting the accuracy of the testing results. Lastly, the generation and execution processes in MBT testing are all dependent on computer algorithms, which means that error omission caused by human factors is isolated from the testing procedure. In MBT testing, the only element which needs to be develop by humans is the formal model of the SUT behaviour which is also known as the test oracle [35]. Provided the formal model is correctly built according to the specification requirements of the SUT, it promises to obtain a convincing testing result which determines whether SUT behaviour complies with the given specification requirements. Overall, MBT is more eligible than manual testing for testing safety-critical systems such as TCSs.

Although MBT has been rapidly developing and has been proven to be suitable for testing large-scale systems including software and hardware, it is still challenging to apply MBT for testing industrial-sized systems with a high degree of complexity, such as TCSs which contain many subsystems and components with complex interactions and many nondeterministic situations. As one of the key steps in MBT methods, formal modelling is relatively difficult compared to manual test generation, especially when dealing with complex modelling subjects such as TCSs. Since formal language does not describe the modelling subjects in a natural manner which can be understood by most people, formal modelling can take longer than manual test case drafting, even for an experienced tester. When the modelling subjects are of industrial size, they can consist of numerous components with intricate structures for realising various functions by series of interactions, which exponentially increase the

difficulties of formal modelling. Even if formal models are constructed successfully, they can still be too complex to be processed by computers in an acceptable time frame. When a formal model becomes too complex, which means that there are too many possibilities contained in it, state explosion may happen when applying test generation and execution algorithms so that the computational resources of computers can be exhausted, which means the MBT cannot be applied to test industrial-sized SUTs without controlling the formal modelling scales. Different from manual testing methods which specifically emphasise sequences of valid inputs and corresponding expected outputs, the modelling methods of MBT model SUTs in a format of different types of formal expression, which is not understandable for nonexperts. Therefore, it is difficult for an inexperienced tester to determine whether the specification model correctly presents every essential element involved in the specification requirements of the SUT. However, without correct formal models, the test cases derived can be invalid or inaccurate for use in testing the SUT, so that the testing results obtained are meaningless.

Compared with MBT methods, manual testing methods are mostly straightforward, conforming to natural human habits of testing, and are understandable for most testers who are familiar with the functional characteristics of the SUT. Unlike formal models, the correctness of which needs to be verified by relevant techniques such as model-checking [36, 37] and theorem proven, test cases for manual testing are drafted by the experts from authoritative organisations such as UNISIG, which takes charge of standardising the Form Fit Functional Interface Specification (FFFIS) of all subsystems and key components contained in the ETCS system, and the corresponding test cases and test facility for those test

specifications. With technical support from the professional company members of UNISIG such as Alstom, Ansaldo, Bombardier, Siemens and Thales, the correctness of the test cases is convincing. That is why the testing of TCSs still depends mainly on manual testing, though it relies more and more on formal methods such as MBT.

1.2 Motivation and Objectives

To address the challenge of applying MBT methods in TCS testing, the limitations of formal modelling methods must firstly be overcome. Therefore, the author has developed a novel MBT method called simulation combined MBT, which overcomes the aforementioned difficulties of utilising the MBT method to test complex systems. In contrast with traditional MBT methods which describe SUT behaviour in a single formal language or in multi-layer formal language, simulation combined MBT obtains the SUT model from formal modelling combined with simulation, targeting two types of system behaviour in two models. To decrease the modelling difficulties as well as to control the model complexity under an acceptable level, the SUT model is divided into two models, the abstract models in charge of abstract and discrete system behaviour, and the simulation model in charge of concrete and continuous system behaviour. Based on the two-model-combined structure, the system behaviour for relatively complex SUTs can be modelled entirely without the risk of state explosion. Furthermore, it simplifies the process of building formal models by moving most of the continuous behaviour, which is difficult to model in formal language, from formal models into simulation models.

Compared with formal language, simulation is more applicable for describing continuous behaviour from a macroscopic view because of the flexible features of simulation. For a complex system with hybrid characteristics, such as a TCS, combining discrete condition-switching, such as the transition mode of the On-Board Unit (OBU), and continuous variable changes, such as the train speed varying in operational procedures, the two-model-combined structure takes advantage of both formal modelling, which is adept at describing discrete transition processes, and of simulation, which is good at depicting continuous variation processes. With the combination of both modelling methods, the modelling difficulties of the SUT and the processing difficulties of the SUT models are together reduced, which significantly increases the feasibility of applying MBT methods in complex system testing. Since the modelling method is different from those of traditional MBT, the test tool which is utilised to generate test sequences based on the analysis of formal models cannot be directly adopted in simulation combined MBT. Therefore, a customised interface has been developed for the application of an online MBT test tool, allowing the online testing of complex systems to be realised.

In the field of MBT, online testing and offline testing are two contrary concepts of different kinds of testing implementation technique; offline testing generates test cases then executes them, while online testing generates and executes test cases simultaneously. Online testing can deal with nondeterminism contained in the formal model, but performs worse than offline testing in checking strict time restrictions because the test cases cannot be generated in time for execution when the formal model is too complex. By comparison, offline testing cannot

deal with nondeterminism contained in the formal model, but it is good at checking strict time restrictions, and the generation and execution processes are separated. Since a TCS is a typical nondeterministic system with a high degree of complexity, the author aims to implement online MBT testing based on the simulation combined MBT method introduced, considering testing accuracy and efficiency, which leads to the following objectives:

- Discuss the main tasks of testing the functions of a TCS.
- Based on the discussion, explore the feasibility of applying MBT to test TCSs.
- According to the exploration results, develop a simulation combined MBT which is suitable for testing TCSs.
- With application of the simulation combined MBT, build up an online MBT testing platform which can be applied to test different types of TCS in various railway networks.
- Implement online testing based on a case study of a TCS utilised in real railway systems and draw an eventual testing verdict.
- Verify the testing results obtained, determine the effectiveness of the simulation combined MBT, analyse whether testing performance is better than that of existing testing methods.

With all the objectives achieved, the author expects that simulation combined MBT can be applied to test TCSs and other industrial-sized systems with complex functions and structures. With verification of the testing results, the online MBT platform developed by the author is expected to obtain better performance in terms of testing correctness, functional coverage and

time efficiency.

1.3 Thesis Structure

The thesis is presented with the following structure:

- Chapter 2:

  Different types of TCSs, which are the SUTs, are introduced. Traditional MBT methods and traditional functional testing of TCSs are introduced. Based on the review, the research problem is formulated.

- Chapter 3:

  The reason for choosing online MBT is explained. Evolved from the traditional modelling method for online MBT, the Simulation Combined Timed I/O Transition System (SCTIOTS) modelling theory is developed with formula derivation. Based on SCTIOTS, simulation combined MBT methodology is proposed.

- Chapter 4:

  The method of implementing simulation combined MBT is introduced by developing a simulation combined MBT platform. The essential components of the platform are introduced, including the modelling tools, test tool, I/O sequence manager and Hardware-in-the-Loop (HIL) environment. The architecture of the platform is explained at the end of the chapter.

- Chapter 5:

  Two case studies are undertaken to prove the feasibility of the proposed testing method

and developed platform. The first case concentrates on explaining the developed components of the simulation combined MBT platform. The second case concentrates on testing the overspeed protection function and the train location function of an SUT Vehicle On-Board Controller (VOBC). Testing results are recorded and analysed in both cases.

- Chapter 6:

  The effectiveness and performance of the simulation combined MBT platform are verified, including validation of the specification requirements, verification of the effectiveness of the testing platform, and verification of the performance of the testing platform. Impact factors of test efficiency and quality are explored at the end of the chapter.

- Chapter 7:

  The conclusion and contribution of the thesis are summarised. Future work is presented at the end of the chapter.

2 Literature Review of Functional Testing in Train Control Systems and Model-Based Testing Methods

Testing is a broad concept with definitions that can vary from field to field, and each one can be quite different from the rest when considering different testing purposes and testing scales. Therefore, before MBT can be applied to test an SUT, three essential elements, the scale of the SUT, the scale of the testing and the purpose of the testing, must be specified to determine the appropriate type of test. Evolved from the model defined by Utting and Legeard [30], the concept of different types of testing is generally defined by Fig 3:

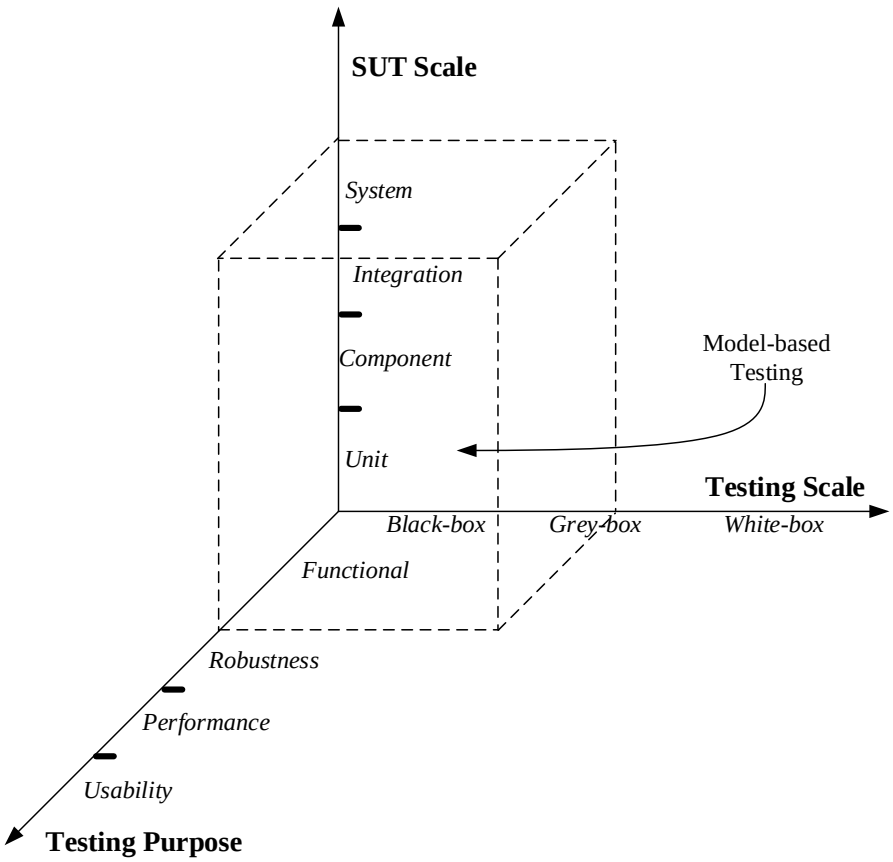


Fig 3 Classification of different types of testing

As depicted in Fig 3, three key indices profile different types of testing by determining the testing purpose, testing scale and SUT scale of testing. For testing scale, black-box testing means that the tester does not have the access required to know the internal behaviour of the SUT, while white-box testing means that the tester does have access to the internal behaviour of the SUT [38]. That is to say, white-box testing aims to test the internal behaviour of the SUT, which means that the tester needs to understand its internal operating principles [39]. In grey-box testing, which is related to black-box testing and white-box testing, the tester only has partial knowledge of the internal SUT behaviour, so that it can have the characteristics of both black-box testing and white-box testing and can be a richer approach [40]. For SUT scale, testing has different meanings when it is implemented at different levels of SUTs, including unit testing, component testing, integration testing and system testing. Obviously, it is difficult to distinguish these four levels in complex systems with a complicated structure, such as TCSs which consist of a series of subsystems, components and units. For such a system, it necessary to define the boundary between the internal and external layers of the SUT, without necessarily defining which level the testing belongs to. In testing purposes which directly determine the testing type, testing is classified into different categories, including functional testing, robustness testing, performance testing and usability testing, each of which refers to corresponding testing methods. As mentioned above, functional testing aims to verify the system's functional behaviour which is designed and developed within the system specification requirements so that it usually connects with black-box testing. As conclusively indicated by Fig 3, the relevant fields of MBT are restricted inside the dotted cube, indicating

that MBT is designed and implemented for functional testing with a black-box or grey-box testing scale, though it can be adopted to test any level of the SUT. It is worth noting that the dotted cube does not mean a strict restriction, which means that the MBT can still be utilised for other testing purposes, such as performance testing and robustness testing. However, functional black-box testing is the main application scenario.

2.1 Introduction to Train Control Systems

As mentioned in section 1.1, different types of TCS are selected for use in different countries based on national rulebooks and other constraints. To indicate that the proposed testing method can be adopted to test ETCS, CBTC or other TCSs, the author has generalised the system structures of the different TCSs to illustrate their similarities.

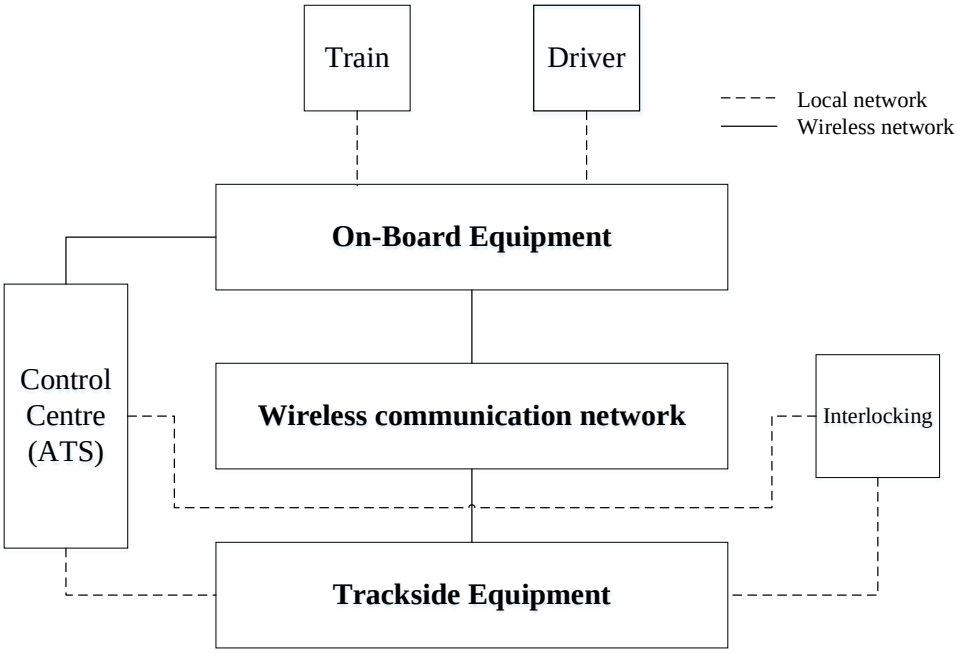


Fig 4 Generalised system structure of ETCS, CBTC or other TCSs

As indicated by Fig 4, different types of TCS share a generalised system structure which includes the on-board equipment, trackside equipment and radio communication network. On-board equipment, such as the OBU in ETCS or the VOBC in CBTC, is responsible for guaranteeing the safe movement of the train and feeds back the operation conditions of the train to the trackside equipment. Trackside equipment, such as the Radio Block Centre (RBC) in ETCS or the Zone Controller (ZC) in CBTC, is responsible for indicating where the train should go along the track and feeds back the track conditions to the control centre or Automatic Train Supervision (ATS), in collaboration with the interlocking. ATS is responsible for supervising the operation conditions of the integrated railway system, and sends macro-control commands when necessary, such as sending a rescheduling command when a delay happens. The on-board equipment is installed on the train to control the train movements. The driver can send a command to the on-board equipment via the Driver–Machine Interface (DMI) when necessary. Based on the generalised structure of different types of TCS, functions need to be realised collaboratively by two or more of the elements presented, which means that the functional behaviour of the systems can be complex, and the modelling difficulties can increase. Even so, TCSs can be modelled in the same framework, which means they can be tested by the same MBT method. The following table shows a comparison of the functions and system composition of ETCS and CBTC systems [41], where ‘X’ denotes the presence of the component and ‘-’ denotes the absence of the component:

ETCS						**CBTC**	
Equipment		Operation Level				Equipment	
		L0	*L1*	*L2*	*L3*		
OBU	DMI	X	X	X	X	VOBC	TOD

	BTM	-	X	X	X		TIA
	TCR	X	X	X	-		-
	ODO	X	X	X	X		ODO
	EVC	X	X	X	X		KVC
	Euroradio	X	X	X	X		OBRU
	JRU	X	X	X	X		DR
	TIU	X	X	X	X		TIMS
Trackside	Eurobalise (or Euroloop)	-	X	X	X	Trackside	Balise
	TC (or axle counter)	X	X	X	-		Axle Counter
	LEU	-	X	-	-		-
	RBC	-	-	X	X		ZC
Wireless network	GSM-R	X	X	X	X	DCS (WLAN)	

Table 2 Comparison of ETCS and CBTC systems

In ETCS and CBTC systems, some components with similar functions are given different names, such as the European Vital Computer (EVC) in ETCS and the Kernel Vital Computer (KVC) in CBTC, which are both vital computers providing the necessary computations for train control. Similarly, Euroradio and On-Board Radio Unit (OBRU) are both radio communication terminals for on-board equipment of ETCS and CBTC. Overall, ETCS and CBTC systems control train movements by the cooperation of on-board and trackside equipment. Bidirectional communication is established between the on-board and trackside equipment to exchange information essential for their operation. Four operational levels are included in ETCS systems to adapt to the operation of the legacy railway systems existing in different European countries.

In order to fulfil the reviewed system requirements, companies such as Siemens, Bombardier,

Thales and the China Railway Rolling Stock Corporation (CRRC), have developed their own CBTC solutions. Although the CBTC systems developed share the same architecture as illustrated in Fig 4, the components used in each of the systems are different; the author has therefore not fully listed the components of the CBTC system, instead including only the main components and subsystems. One prominent feature of CBTC systems is that Wireless Local Area Networks (WLAN) are most commonly used as the radio communication network of the Data Communication System (DCS), while ETCS systems usually utilise GSM-R. Due to the different application scenarios, the specific functional performance of the subsystems and components in ETCS and CBTC systems can be different. Nevertheless, the macro-system architecture, specifically the cooperation of trackside and on-board subsystems by wireless communication via a radio network, is highly uniform.

As revealed by Fig 4 and Table 2, ETCS and CBTC systems, which between them represent the majority of modern TCSs, share a united system structure and functional features. For black-box testing which aims to verify that the functions developed comply with the specification requirements, the similarities between the TCSs means that the I/O interface between the SUT and the testing tool can be used for testing different systems, with minor modifications. For HIL testing, similar system structures and composition mean that the simulation for an HIL environment can be used repeatedly without major modifications when testing different types of TCS. For MBT, formal modelling of TCSs with similar features means that modelling difficulties will not increase when testing different types of TCS. As a result, functional testing of modern TCSs based on a unified testing method can be

promisingly realised.

2.2 Traditional MBT Methods

Based on the discussion in section 1.1, MBT can significantly reduce the cost and time associated with testing, and achieve better quality performance by having better traceability and extendibility compared with traditional manual testing methods. Despite this, it is not a flawless testing method without limitations. Firstly, MBT does not enhance the ability to detect defects in the SUT, because it still relies the tester to build the specification model, which is the mechanism used to determine whether the test should pass, and to choose the test generation strategies; this means the performance of the MBT is determined by the skill and experience of the tester [30]. Further, MBT cannot be guaranteed to find all the errors contained in the SUT, which is the limitation for all other testing methods [26]. Nevertheless, with a well-modelled test oracle and a correctly selected test generation strategy, MBT increases the possibility of finding errors at a lower test cost and in a shorter test time. This leads to the second disadvantage of MBT, which is that it is more difficult to implement than manual testing because of difficulties in formal modelling and test generation algorithms [18]. To formalise the SUT behaviour, MBT demands that testers have a deep understanding of the operation principles of the SUT so that they can build a precise and unambiguous model which can be understood and processed by various test tools. It can take the tester years of practice to be familiar with one type of formal modelling method and the corresponding test generation algorithms. Furthermore, testers should understand how to test SUTs manually in order to build specification model for MBT because MBT is an automation of manual testing

methods.

In addition, MBT is strongly associated with functional testing and is rarely utilised in other types of testing, except that it is occasionally used in stress testing [30]. Some kinds of SUT, such as those which involve plenty of man–machine interactions, are not eligible for applying MBT because of their unique characteristics. For example, the DMI which is one of the components in the OBU is not suitable for automatic testing methods such as MBT because it is designed to provide driving instructions for the driver via a screen. Although MBT can check whether the I/O data of the DMI is correct, it cannot prove that the corresponding screen display is correct. As a result, these kinds of SUT should be manually tested. Even worse, analysis of failed tests in MBT can be time-consuming because the testing results obtained are in a formal format, which can be understood by a computer but is not convenient for inexperienced people to understand. By comparison, testing results in manual black-box testing are easier to analyse because they are straightforward so that the tester can locate the errors by comparing the results with the specification requirements.

Lastly, traditional MBT methods mostly require that the specification model is deterministic regardless of whether the SUT itself is deterministic. However, complex systems such as TCSs can be nondeterministic in some or all layers, including the unit layer, component layer, subsystem layer and system layer. Elimination of nondeterminism is not only a time-consuming and difficult process but can also be a risky operation leading to state explosion. Online MBT algorithms are designed to deal with nondeterministic SUTs, which further increases the difficulties in implementing MBT. As mentioned in section 1.1, online

testing is more appropriate for the implementation of TCS testing due to its ability to deal with nondeterminisms contained in the system with less strict time constraints. Since online testing needs to simultaneously generate and execute test cases, the test tools must be highly synchronised with the SUT to guarantee that the observed I/O sequences are valid for the defined outcome criteria. Therefore, the interfaces used for mapping the abstract I/O for test tools and real I/O for SUTs are one of the key elements in testing implementation, and any wrong I/O mapping or poor efficiency of translation can lead to a failure result. Another adverse factor is that a communication delay between the test tool and the SUT becomes non-negligible when the SUT is a timed system with a set of time constraints [42], especially when dealing with complex SUTs such as TCSs which include communications between hardware and software components. This increases the modelling difficulties in online MBT. Because of the difficulties and limitations mentioned, online MBT is only supported by a few MBT tools, such as QTronic and UPPAAL-TRON.

Despite the limitations discussed, online MBT is still a feasible solution for automatic testing. Unlike other automatic testing methods such as script testing or keyword-driven testing, which incompletely automate the functional testing process, MBT can completely automate the test process: the test is automatically generated and executed, and the testing results are automatically qualitatively analysed. Therefore, MBT is adopted more and more to save testing time and resources, improve testing quality and guaranteeing its correctness.

For MBT, modelling methods, test selection criteria and test tools are the three key elements, and the tester needs to make appropriate choices for each of these three according to the

characteristics of the SUT and its testing environment. Hundreds of different modelling methods have been used to describe SUT behaviour for MBT. As one modelling method commonly corresponds to one or more test generation tools, the author will first classify and introduce different MBT methods by introducing various modelling methods. Next, the author lists a series of test selection criteria for different types of model with different testing purposes. Finally, the author introduces several test tools which support the modelling methods and test selection criteria introduced.

2.2.1 Introduction of Modelling Methods for MBT

As a wide-ranging concept which can appear in many fields, modelling has a set of different meanings for different purposes. Since MBT mainly depends on formal modelling methods which can be used as the test oracle for test case generation and test result verdicts, the author focuses on formal modelling methods in this thesis [43]. The basis for classification can vary from person to person, and the author has adopted one proposed by Utting and Legeard [30]. The original classification targets the industrial field, introducing almost every modelling method involved with MBT in detail. In this thesis, the author has refined the original classification scheme by omitting modelling methods which are rarely utilised in MBT, and explains in detail those which are commonly adopted for research purposes. As a result, the modelling methods for MBT are divided into three categories; state-based modelling methods, transition-based modelling methods and other modelling methods.

2.2.1.1 State-Based Modelling Methods

State-based, also known as pre/post, modelling methods depict a system based on a set of states with variables in and constraints on those states. In one state, actions or operations may happen when the corresponding conditions are satisfied, and the variables are then updated according to the defined relations. State-based modelling methods concentrate on describing the internal conditions in states, and therefore weaken the external transitions between two states. As a result, they are more suitable for modelling data flow-oriented SUTs of which functional testing emphasises correct data flow and is less concerned about control flow. Typical examples of state-based modelling methods include but are not limited to B/Event-B [44, 45], Z [46], Unified Modelling Language (UML), Object Constraint Language (OCL), Java Modelling Language (JML) [47], Spec# [48] and the Vienna Development Method (VDM) [30]. As one of the most typical state-based modelling methods, Event-B will now be introduced by the author with a modelling example.

As an evolved version of the B method, Event-B makes it easier to perform refinement and verification processes with the help of developed software platforms [49]. Summarised by Cansell and Mery [45], the key elements of an Event-B model are illustrated by Fig 5:

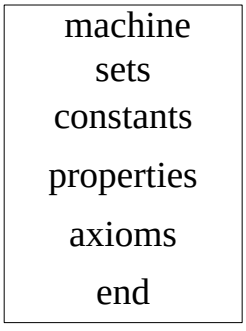


Fig 5 Elements of an Event-B model

As illustrated by Fig 5, Event-B is a contextual modelling notation in which a model consists of the following clauses: the *machine*, the *sets*, the *constants*, the *properties* and the *axioms*. The clause *machine* gives the model a name; the clause *sets* contains definitions of sets in the problem; the clause *constants* summarises the variables involved in the clause *properties* which are the detailed definition of the *sets*; the clause *axioms* contains the invariant rules that should be held by the developed model and which are going to be verified by the proof engine. Once the specification model of the SUT is obtained based on Event-B, the test cases can be derived from the specification model with the assistance of test tools along with selected test selection criteria.

2.2.1.2 Transition-Based Modelling Methods

Compared with state-based modelling methods, transition-based modelling methods emphasise transitions from state to state and concentrate less on the profiles of internal states. One of the typical representatives is the Finite State Machine (FSM), which is a graphical notation describing a system with the pattern node–transition–node. The node represents the essential states of a system, and the transition represents the actions or operations which happen when the transitions happen, as shown by Fig 6:

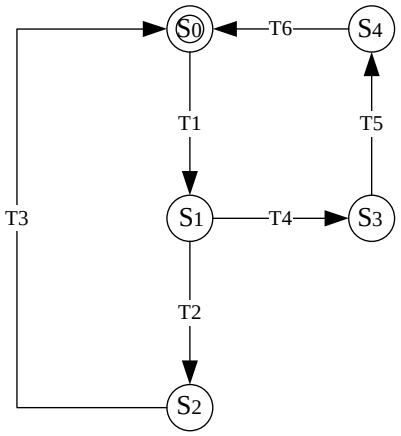


Fig 6 Schematic of finite state machines

In practice, an FSM can be extended by adding snapshots of each state, configuring hierarchical structures for different layers, and establishing parallel connections between several state machines. Extended versions adapt to different characteristics of systems so that they can model the transition flow of the systems without losing other essential system information. Typical examples of transition-based modelling methods include FSM [50] and its varieties such as Labelled Transition Systems (LTS), I/O automata, Timed Automata (TA) and hybrid automata, and statecharts such as UML State Machines, STATEMATE statecharts and Simulink Stateflow charts [30]. Although the methods mentioned have specialisation use in particular scenarios, they share the common points that they are all transition-based; the main differences come from the different configurations of their platform. Therefore, the author introduces two of the methods to indicate the similarities and differences between different transition-based modelling methods.

- Timed automata [51]

As one of the varieties of FSM, TA evolves with a finite set of timed clocks which linearly

increase in states during the operation procedure. TA is suitable for modelling timed systems with linear time constraints. With the assistance of model checkers, TA models can be verified against formalised properties such as liveness which means some states are reachable and safety. The author will now introduce the UPPAAL platform, which models a system based on the TA format [52, 53].

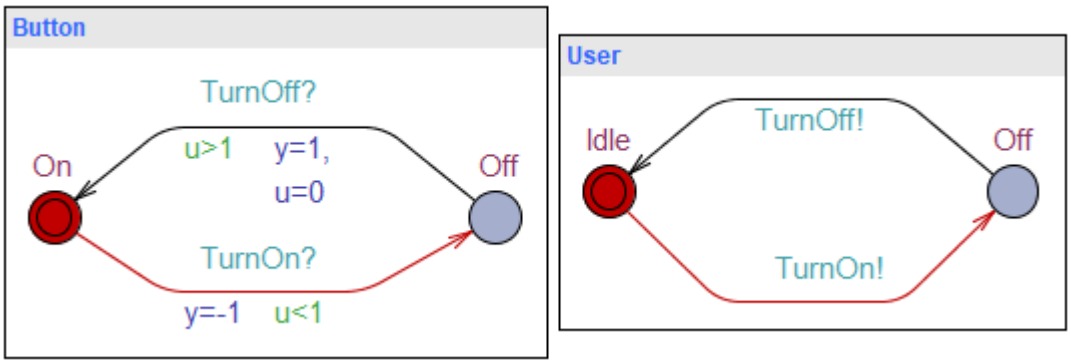


Fig 7 Schematic of TA model on the UPPAAL platform

As indicated by Fig 7 which is a formal model of a button, the TA model on the UPPAAL platform absorbs the features of labelled transitions systems and I/O automata, and the input and output are respectively indicated by specific labels. This configuration is specially designed so that it is more convenient for the test generation tool to recognise inputs and outputs. Another feature is that the TA modelling method models the SUT and its operational environment or user in a parallel structure of two or several automata, where two transitions are synchronised by an input/output pair to happen at the same time. With the corresponding test generation tools, the test cases or sequences can be derived from the TA models built.

- Statecharts

Statecharts are quite similar to FSM-based modelling methods such as the TA modelling

method. However, there are still some differences between them so that the tester should select the appropriate modelling method according to their specific testing purpose. As revealed by Fig 8 and Fig 9, which show a button model built on the MATLAB Simulink platform, statecharts on the Simulink platform can describe systems in a hierarchical structure, while the TA on the UPPAAL platform can only support a parallel structure. This could be an advantage when modelling complex systems consisting of numerous layers. Furthermore, with a more advanced graphical user interface, modelling difficulties can be reduced so that the modelling efficiency can be improved.

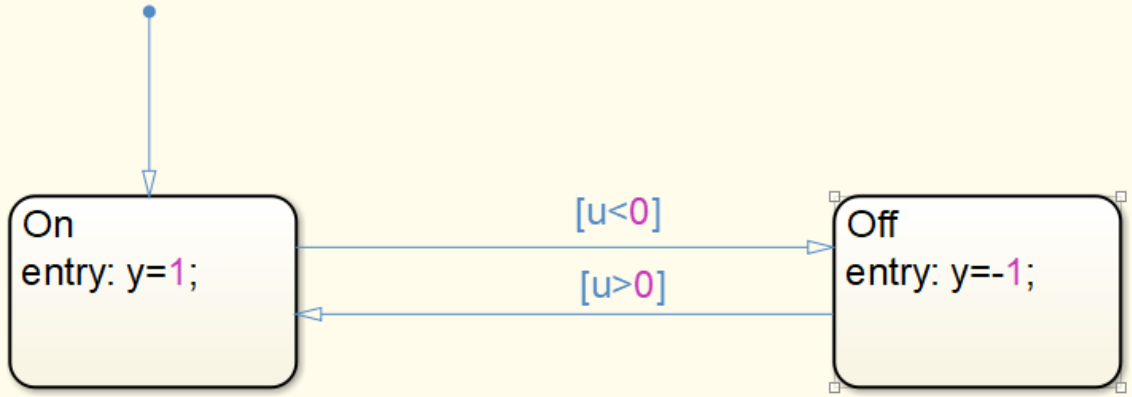


Fig 8: Schematic of statechart model on the Simulink platform

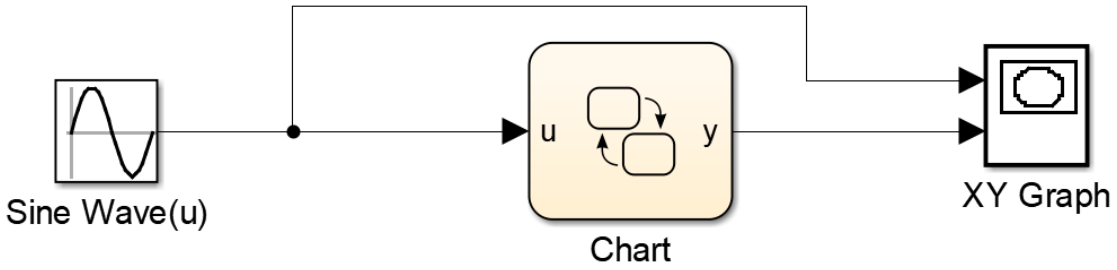


Fig 9 Complete model for the statechart model

As shown in Fig 9, the external stimulus, which can be a continuous or discrete signal, can be

freely defined by the user; this is difficult to achieve with FSM-based tools such as UPPAAL. Based on more simulation models, Simulink statecharts can include more detailed information in the model. However, most of the matched test tools for Simulink statecharts stay at the stage of script testing or keyword-driven testing, both of which entirely or partially rely on the tester to design the test cases; neither of them provides genuine automatic test case generation. According to Fig 2, the efficiency advantage of MBT appears gradually along with an increasing number of test cases, in other words, with the complexity of the SUT. When testing systems like TCSs, automatic design of functional testing becomes more important. Although some of test tools are claimed to be able to realise automatic test case generation, they are still in immature forms. For example, Li and Kumar [54] developed an algorithm for automatic test case generation based on the Simulink statechart model that translated the statechart model into IO-EFA, a variety of FSM, then applied model-checking to generate test cases. The company T-Vec [55] claims that their product can automatically generate test cases, but the test tool is commercial making its performance difficult to verify. According to Blackburn et al. [56-58], their product aims to automatically generate test vectors, a set of various inputs desired for certain testing purposes, which belong to the fields of script testing and keyword-driven (table-driven) testing. In general, most of the test tools based on Simulink cannot realise automatic test case generation.

2.2.1.3 Other Modelling Methods

The author has gathered the remaining kinds of modelling method into this category because they are usually utilised in combination with one or several other modelling methods.

History-based modelling methods such as Message-Sequence Charts (MSC) [59] describe system behaviour by recording the message exchange process between two or more components. As a result, though it can be eligible for modelling communication protocols, the preferred usage is to present the test cases or test sequences generated by specifying the data exchanged between the tester and the SUT. As typical representatives of operational modelling methods which focus on describing interactions between concurrent systems [60], Communicating Sequential Processes (CSP) and Petri Nets (PN) [61] are often used in combination with other modelling methods, such as FSM-based methods, to formally obtain hierarchical system models [62, 63]. Other modelling methods such as functional notations, statistical notations such as Markov chains [64], and data-flow notations are occasionally adopted by some modelling tools for system modelling, verification and testing.

2.2.1.4 Summary

Countless modelling methods have been applied to the MBT field, and it would not be appropriate for the author to include all of them in this thesis. Instead, those most representative of the main categories have been introduced by the author. To automate the testing process, a specification model needs to be utilised to formally describe the SUT behaviour so that the computer can generate test cases by analysing the specification model. Therefore, choosing the appropriate modelling method is an essential foundation of successful implementation of MBT. One of the guiding principles of choosing a modelling method for MBT is to choose according to the characteristics [30] and emphasis of the SUT. For testing data-oriented SUTs where the tester focuses on the key parameters, it is recommended to

choose state-based modelling methods because methods such as the B method, as these support a wide range of data types allowing the tester to precisely describe the SUT behaviour. Transition-based modelling methods, such as LTS, can conveniently describe the complex transition relations between different nodes of the state machine based on the node–transition–node format. However, state–based modelling methods have to specify the precondition and postcondition for every state so that the model becomes unnecessarily large when the transition relationship is complex. Therefore, for testing control-oriented SUTs where the tester is concerned about the transition flow of the SUT, it is recommended to utilise transition-based modelling methods to guarantee modelling efficiency.

The classification between data-oriented and control-oriented systems becomes ambiguous when the SUT is an integrated system with a relatively high degree of complexity, such as a TCS, for which extensive data verification and control-flow verification are both required in the testing. Therefore, the tester should select a suitable modelling method that can fulfil the requirements of testing implementation for such systems. State-based modelling methods can still deal with control-oriented SUTs, and transition-based modelling methods can still deal with data-oriented SUTs. As a result, the type of SUT is not the only basis for determining the modelling method. In conclusion, a modelling method is appropriate if it can precisely describe the SUT behaviour. In Table 3, some modelling methods which have been frequently adopted for MBT are listed with their classification and a brief description.

Notation	Classification	Remarks
B	State-based	Abstract machine notation
Z	State-based	Based on first-order predicate logic and set theory
JML	State-based	Behavioural specification language
Spec# [65]	State-based	Object-oriented language, extension of C#
SeC (C++) [66, 67]	State-based	Applying contract approach, based on C, C# and Java
OCL [30, 68]	State-based	Object-oriented language supporting UML
VDM [69, 70]	State-based	Object-oriented specification language
Statecharts [71]	Transition-based	Formal realisation of FSM
UML SM [72, 73]	Transition-based	Behaviour description language based on UML
Stateflow charts [74, 75]	Transition-based	Supported by UML and MATLAB Simulink
LTS [76, 77]	Transition-based	Behaviour description language, basis of I/O automata and other FSM-based varieties
TA [51, 78]	Transition-based	Extended LTS with time constraints, supported by UPPAAL
MSC	History-based	Often combined with SDL
HOL [79]	Functional notation	Often combined with other software tools
CSP	Operational notation	Often combined with PN
CCS	Operational notation	Often combined with PN
Petri net	Operational notation	Often combined with CSP, CCS
Markov chains	Statistical notation	Good at describing a choice of input, weak at predicting expected output. So, needs to

		be combined with other modelling methods
Lustre [80, 81]	Data-flow notation	Describes concurrent systems, supported by MATLAB Simulink and SCADE
Block diagram	Data-flow notation	For modelling continuous systems

Table 3 Summary of formal modelling methods

2.2.2 Introduction of Test Selection Criteria

With the SUT formally modelled, the next step is to generate test cases from the formal models obtained. Since the formal model can be complex when modelling an industrial-sized SUT, it can be difficult to generate a set of test cases covering all the possibilities contained in the model, which means that the test generation process should be controlled based on a particularly emphasised field which is determined by the testing purpose or the specification requirements of the SUT. Test selection criteria guide the controlling process and are employed by the tester to measure the adequacy of the package of the test cases generated [82]. Given a specified criterion, the test generation tool has guidance on when to stop the generation process and how well the test cases have been generated. Although the ultimate goal of test generation is to generate a test suite which can fully cover the possibilities contained in the formal model, 100% coverage can be difficult to achieve [83]. Therefore, test selection criteria can give the tester an intuitive impression of the test generation performance by measuring what percentage of the requested coverage have been satisfied. Furthermore, during the test generation process, some test tools can cater to the given test selection criteria by applying corresponding test generation strategies so that unnecessary test cases can be omitted, and the test generation resources can be economised. This is extremely important for

test generation from complex models because full coverage can be difficult to achieve in such models, so the tester needs to know whether the test cases obtained are sufficient for functional testing.

Depending on the modelling method chosen, selection criteria can become different concepts. Therefore, the modelling notations should be determined before any discussion of test selection criteria [84]. Since the author's research is strongly related to FSM-based modelling methods, the modelling notations which correspond to the test selection criteria being discussed have been determined to be transition-based modelling methods and modelling methods which can be transformed into a transition-based format, such as state-based modelling methods. As a result, other test selection criteria are not introduced.

Since there has been rapid development of MBT technologies recently, more and more refined test selection criteria are being proposed for specific testing purposes. As a result, the criteria for transition-based modelling methods comprise a large set of concepts including many branch criteria. Here, the author introduces the main kinds of criterion which are typically adopted in MBT. In transition-based modelling methods, the SUT is required to be modelled as an FSM which contains states and transitions. Although different modelling structures, such as hierarchical structure and parallel structure, can be realised by different transition-based modelling methods, the models can always be transformed into one FSM or an approximation of an FSM, as shown by Fig 10:

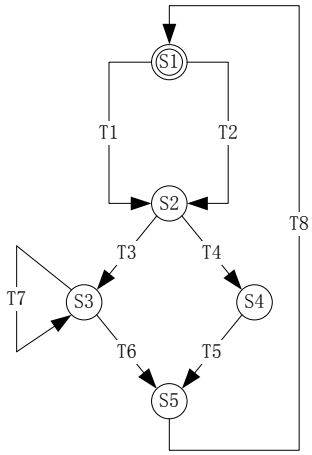


Fig 10 Schematic of finite state machines

As revealed by Fig 10, an FSM consists of states and transitions, where $\{S_1, S_2, S_3, S_4, S_5\}$ presents all reachable states; the double circle is the initial state of the FSM, and $\{T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8\}$ stand for all accessible transitions. Based on the defined conditions, the test selection criteria are discussed in the following sub-sections:

2.2.2.1 All-State Coverage

All-state coverage requires that all reachable states of the FSM, which are $\{S_1, S_2, S_3, S_4, S_5\}$ in the case of Fig 10, should happen at least once in the test cases generated. It should be noted that, when applied to hierarchical structure and parallel structure, all-state coverage may have different meanings. In hierarchical structure, the states are divided into external states and internal states, and coverage of external states does not mean that all internal states are covered, so the hierarchical structure of the model needs to be transformed into an FSM format indicated by Fig 10 before proving all-state coverage. In parallel structures, two transitions can happen simultaneously so that more than one state can be activated at the same

time. Therefore, covering one of the states which are occupied at the same time along with all the other normal states is adequate for proving all-state coverage.

2.2.2.2 All-Transition Coverage

All-transition coverage requires that all accessible transitions, which are $\{T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8\}$, should happen at least once in the test cases generated. Similar to all-state coverage, when applied to hierarchical structure and parallel structure, all-transition coverage may have different meanings. In hierarchical structure, transitions between external states and transitions between internal states should all be covered. Therefore, the hierarchical structure of the model needs to be transformed into an FSM format indicated by Fig 10 before proving all-transition coverage. On the other hand, in parallel structures, two transitions can happen simultaneously so that more than one transition can be accessed at the same time. Therefore, covering one of the transitions which happen together along with all the other normal transitions is adequate for proving all-state coverage. All-transition coverage is a stronger criterion than all-state coverage, which means if all-transition coverage is achieved, all-state coverage will be always satisfied.

2.2.2.3 All-Path Coverage

In an FSM, a path is a sequence of states and transitions leading to a certain state. In an FSM, all-path coverage requires that all valid paths should happen in the test cases generated, which can be difficult to achieve because an FSM can contain an infinite number of paths. For example, in Fig 10, the paths are countable without transition ‘T7’ but are infinite with

transition ‘T7’ because the number of times that ‘T7’ happens is nondeterministic. Therefore, with a model which is more complex than the one in Fig 10, it is difficult to find out all valid paths and to cover them. All-path coverage is the strongest criterion, which means if all-path coverage is achieved, all-state coverage and all-transition coverage will be always satisfied.

2.2.2.4 All Definition-Use Coverage

In some varieties of FSM such as LTS and I/O automata, variables can be defined and used by the model expressions so that the variable values can be updated along with some of the transitions in the model. Definition-use coverage requires that all paths defining and executing all variables should be covered in the test cases generated. All definition-use coverage can be adopted in test generation for data-oriented SUTs to exhaustively inspect that all data-related operations are functionally correct.

2.2.2.5 Summary

Due to the diversity of modelling methods for MBT, test selection criteria are closely associated with specific modelling methods and are not compatible for other modelling methods. Since simulation combined MBT is based on FSM-based modelling methods, the author has only introduced the relevant test criteria and has omitted others. It should be noted that the criteria discussed can only be utilised for offline test generation where the test tool can record the states occupied, transitions triggered and variables executed during the whole testing procedure. For online test generation which randomly selects and verifies one of the valid inputs; the purpose of the criteria is to verify the test results by measuring what

percentage of the coverage expected has been covered in the testing. The detailed verification procedure is presented in Chapter 6 – Validation and Verification.

2.2.3 Introduction of Test Tools

In recent years, more and more test tools have been developed for automatic test generation in which test cases are algorithmically derived from specification models. Selection of tools is commonly based on the purpose of the testing, the characteristics of the SUT, and the tester's maturity level for different modelling methods. For example, testing of an SUT with time constraints requires that the specification model supports the formal expression of a timed operation. In other words, the test tools should be determined by the modelling methods which are appropriate for modelling the SUTs. A test generation tool can support a single format of models or a set of similar types of model. Therefore, the author lists the test tools along with the modelling methods they support, whether they are for commercial or academic use, and whether or not they support online test generation.

Name	Modelling notation	Commercial/academic	Offline/online mode
T-Vec	Simulink, MATRIX	Commercial	Offline
QTronic	TTCN-3, UML	Commercial	Online/Offline
LTG	B, UML 2.0	Commercial	Offline
Reactis	Stateflow (Simulink)	Commercial	Offline
TAU Tester	TTCN-3	Commercial	Offline
Spec Explorer	C#	Microsoft	Offline/Online
UPPAAL-TRON	I/O automata	Academic	Online
UPPAAL-COVER	I/O automata	Academic	Offline
Torx [20]	SDL	Academic	Offline
ASML	XML, Word	Academic	Offline
MulSaw	JML	Academic	Offline

Table 4 Summary of test tools for model-based testing [85]

As indicated by Table 4, only QTronic and Spec Explorer [65] can switch between online and offline modes, and can deal with nondeterministic models by implementing online test generation. UPPAAL-TRON [86] is the only academic tool which supports online test generation, and it cannot switch to offline mode. It is worth emphasising that test tools should service the modelling methods and should be selected depending on the specific requirements of the test. With a wisely chosen modelling method and the corresponding test generation tool, a tester can automate the design process of functional testing under the control of the test selection criteria determined.

2.3 Functional Testing for Train Control Systems

According to Fig 3, functional testing for TCSs commonly belongs to black-box testing, aiming to verify that the functions are correctly developed based on the system specifications. Nondeterminism can be observed during the procedure of black-box testing because the internal actions of the SUT are inaccessible or the testing environment are too complex to be determined. HIL and MBT have been introduced into the field of TCS testing to improve testing performance and reduce testing cost.

2.3.1 Hardware-in-the-Loop testing for TCSs

Since TCSs are highly integrated and complex systems containing many subsystems and components, functional testing of TCSs involves a wide range of different types of testing for different testing purposes and various SUTs. These different kinds of testing are implemented

at different development stages of the TCS, by those with different roles involved in the whole development procedure, such as the product manufacturer and the third-party tester. Therefore, a standard is needed to classify the specific testing responsibilities of every role at every development stage, as shown in Fig 11 [87]:

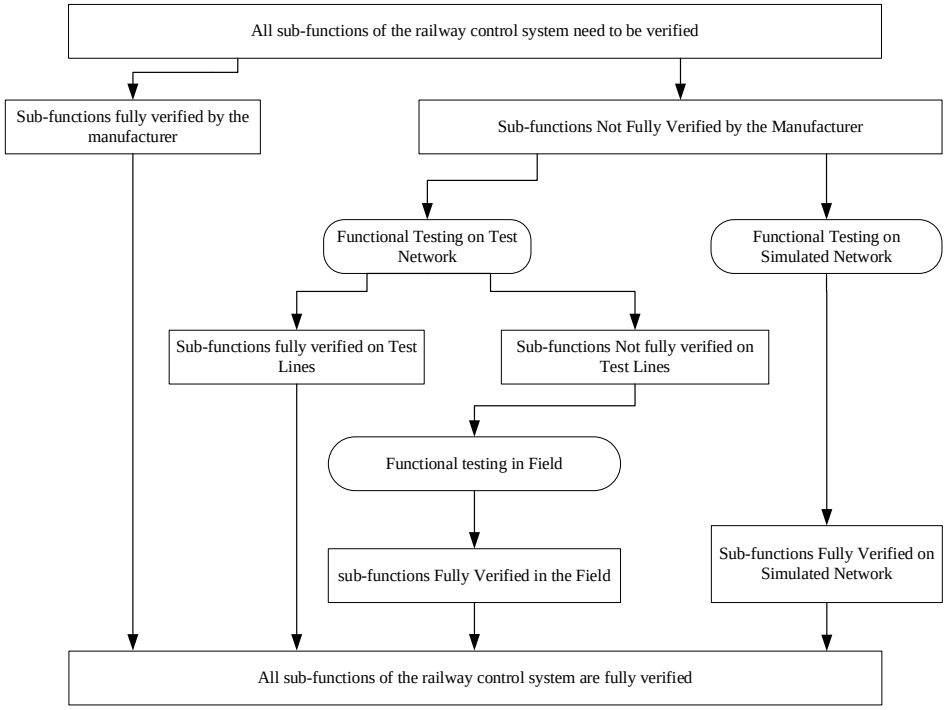


Fig 11 Classification of the testing process by different stages of system development

As shown in Fig 11, the IEEE recommends that those with different roles in development of the system implement functional testing at different development stages to test different system functions [88]. Some of the functions should be tested by the product manufacturer before the subsystems or components are handed over to the third-party tester; this refers to the unit testing and component testing mentioned in Fig 3. It is more convenient and convincing for all such kinds of testing to be implemented by the manufacturers of SUTs because they are more familiar with the internal behaviour of the SUT than testers from

different departments or even different companies. After the internal behaviour of the SUT is verified, the SUT is then ready to be functionally tested in different types of testing scenario, including testing on test lines and testing on real lines. However, due to the high degree of complexity of TCSs, a lot of system functions need to be realised by more than one subsystem or component, which means site testing such as testing on testing lines or real lines can only be implemented after all the subsystems or components involved are ready. In the development of TCSs, it is common that different subsystems and components are developed separately and have different development periods. Therefore, off-site testing is necessary for the system developer to verify the developed part of the system as early as possible [89].

The HIL testing method, which is illustrated in Fig 11, is a feasible solution to achieve off-site testing. Once development of a subsystem or component of the TCS is finished, it can be functionally tested in the HIL testing environment where all the other necessary subsystems, components and network infrastructure are simulated. Therefore, on the simulated network, several key subsystems such as the OBU, RBC, Computer-Based Interlocking (CBI) and ATS can be respectively tested in parallel then tested when integrated, which saves a lot of time. This configuration decreases the chances of damaging SUTs compared with site testing and increases the likelihood of locating errors because the testing scale is limited to subsystem or component level. However, it requires that the simulated HIL environment should be as comprehensive a copy as possible of the real network so that the SUT can operate as it would in a real network. The simulated HIL environment can be provided by a simulator which simulates all essential components in TCSs, such as infrastructure, vehicles, signalling, ATS,

etc. As more and more accurate simulation technologies are applied in the railway field, railway simulation is approximating real railway systems so that more and more functional testing can be realised by HIL testing which can be implemented in off-site scenarios. For example, in UNISIG Subset-094-0 [90], the functional requirements for an on-board ETCS test facility are standardised, and an HIL testing platform is accordingly established by Fig 12. As revealed by Fig 12, the HIL testing platform for the OBU consists of two main parts, the equipment under test and the test environment. The equipment under test contains the components of the OBU and the corresponding adapters. The test environment includes all the other simulated subsystems and components which are necessary to realise the OBU functions, such as lineside equipment and the communication protocol. During the testing procedure, all the equipment of the OBU under test works together with the testing environment by exchanging relevant data via the external communication channels. The condition of the testing environment influences the control command sent by the OBU and vice versa. By monitoring the data flows for each component of the OBU, the tester can judge what has happened in the testing procedure, and whether the test is passed according to the expected data flows derived from the test cases.

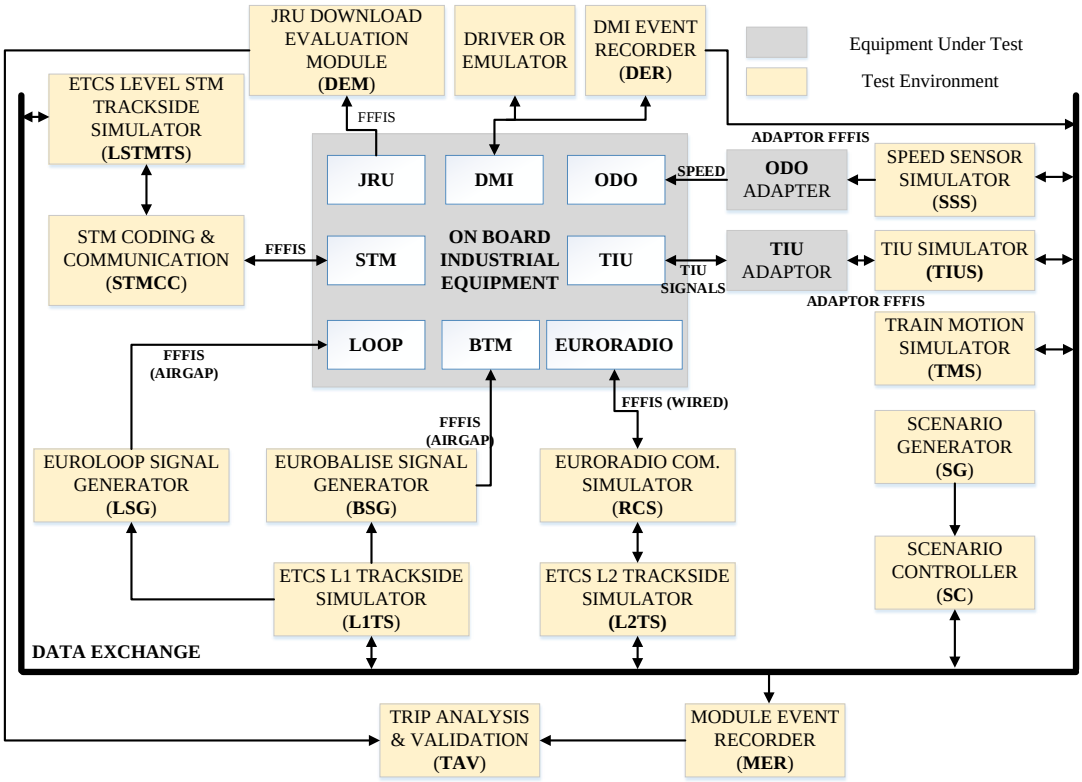


Fig 12 HIL testing platform for the OBU of CBTC systems [90]

Obviously, HIL testing which tests SUTs within a simulated environment is more convenient than testing them on a real site, which means that most functional testing can be implemented in the laboratory environment. However, the testing is still undertaken by a traditional black-box testing method which means that the test cases are manually written, and test sequences are manually derived from the test cases. Although test execution can be automated by scripts, the efficiency of HIL testing is largely influenced by human factors, so that the workload is still heavy compared with automatic testing methods such as MBT. When testing complex SUTs such as TCSs, the functional testing for a single subsystem such as an OBU contains hundreds of test cases, let alone the full functional testing for all subsystems in a TCS. As mentioned in Fig 2, the duration of manual testing significantly increases with the

number of test cases, while the duration of MBT has a much lower growth rate along with the number of test cases. Therefore, to optimise the performance of functional testing for TCSs, HIL testing is merely the first step, and a series of improvements need to be achieved.

2.3.2 Model-Based Testing for TCSs

Due to the safety critical characteristic of modern TCSs, model-based approaches are widely applied to guarantee system safety in system design, verification and testing. Cullyer and Wong [91] combine HOL mentioned in Table 3 with the programming language Ada to automatically verify the signalling design on a given layout of railway network. By formally modelling the railway interlocking table designed for a junction and analysing the obtained formal model with software, the interlocking system is verified to inspect whether there exist any flaws leading to dangerous situations. Piccolo et al. [92] develops a customised formal modelling method for TCSs which can formally represent system behaviour in statechart diagram according to system specification requirements. By processing the formal model by software, system behaviour can be formally verified and test cases can be automatically generated. Dincel, Eris and Kurtulan [21] propose a systematic solution for model-based development of railway signalling and interlocking. With the assistance of model-based techniques, the control logic of the system is designed, verified and refined at system development stage, which significantly improves the system development efficiency and decrease safety flaws. Further research on the formal verification of safety critical components of TCSs are carried on by Ghosh et al. [93]. They develop a bounded model checking algorithm which can deal with a larger scale of signalling and interlocking systems

with a higher degree of complexity, which is helpful for global verification of an industrial sized system with a better flaw detection ability. Ding, Jiang, and Zhou [61] apply Petri Net to formalise the system specification requirements in natural language to eliminate potential ambiguity existing in the requirements, which is meaningful for improving the correctness of system description.

Except system verification, formal methods are applied to undertake MBT in TCSs. Lv et al. [62] propose a layered modelling theory which adopts CSP and UPPAAL as the two modelling methods. Based on the obtained model, test cases of a SUT is automatically generated and coverage of the generated test cases is analysed. Wei Zhang et al. [82] discuss the optimal strategy of test generation for testing the function of MA handover between two adjacent RBCs. All-path coverage is achieved by the adopted test generation algorithms with different strategies and generation efficiencies. Chai et al. [94] propose a framework for runtime verification of TCSs of ETCS. With an integrated formal model of the system behaviour, their verification algorithm can determine whether the system behaviour complies with the specification requirements during system operation.

2.3.3 Summary

According to 2.3.1 and 2.3.2, HIL testing and MBT technologies partially resolve the challenges in functional testing of TCSs. HIL testing decomposes complex TCSs and reduces SUT complexity. MBT technologies automate system verification and testing with a better performance on efficiency and functional coverage. Therefore, a successful combination of HIL and MBT could be an effective solution of the challenges existing in the functional

testing of TCSs.

2.4 Research Problem Description

According to the reviewed background and literature, the existing research on functional testing approaches for TCSs and traditional MBT methods has the following outstanding problems:

- TCSs are too complicate to be fully modelled in formal language;
- The existing MBT technologies cannot process highly complex model due to state explosion;
- Functional coverage of HIL testing cannot be guaranteed because test cases are manually designed in HIL testing.

Therefore, the author of this thesis expects to address the following questions:

- Can the modelling difficulties for complex SUTs be reduced?
- Can state explosion be avoided in the implementation of MBT?
- Can the functional coverage of HIL testing be improved?

According to the problem description, a simulation combined MBT methodology is proposed in this thesis. The modelling and implementation methods are explained in Chapter 3 and 4 respectively. Chapter 5 undertake two case study to explore whether the proposed simulation combined MBT is suitable to test functions of TCSs. Chapter 6 verifies the effectiveness and performance of the developed testing platform based on the obtained testing results, proving that the proposed simulation combined MBT can guarantee a better functional coverage

comparing with existing testing methods.

3 Modelling for Simulation Combined MBT

3.1 Comparison of Online MBT and Offline MBT

3.1.1 Overview of Online MBT and Offline MBT

In Chapter 2.2, the author introduced different MBT methods in terms of different formal modelling methods, various test selection criteria and the existing test generation tools. According to the discussion, most of the methods introduced are offline testing methods which successively generate and execute test cases based on the specification model and test selection criteria [62, 95]. With the assistance of test generation tools, coverage of the test cases generated can be measured so that coverage performance can be improved in offline MBT. However, offline MBT requires that every input must correspond with only one output, which means it is necessary to obtain deterministic models when testing nondeterministic SUTs of which inputs and outputs do not have one-to-one correspondence [96]. Nondeterminism can be observed in black-box testing because of uncertainty of communication delays between test tools and SUTs and lacking details in abstract models. The transformation from nondeterminism to determinism is not only time-consuming but also carries a risk of state explosion. Even worse, some of the nondeterministic models are difficult to transform into deterministic models. To solve the conflict between MBT and nondeterminism, online MBT has been developed to realise automatic test generation based on nondeterministic models.

As a solution for test generation based on nondeterministic models in MBT, online MBT

randomly generates one of the valid inputs from the specification model then executes it and compares the result obtained with the expected one [97, 98]. Due to the online feature, it is able to deal with nondeterministic or highly complex SUTs because it is not limited by the size of the specification model [99]. Within the testing time, online MBT exhaustively searches for all possibilities by randomly generating valid inputs, leading to the limitation that it cannot positively guarantee that all the possibilities can be covered in the testing. The testing verdict is determined by whether inconsistency can be found within the defined testing time, where '***Pass***' means no inconsistency is found, and '***Fail***' means inconsistency is found during the testing process. In contrast to offline testing, online testing generates inputs according to the next reachable set of states, without the influence of other test selection criteria adopted in offline testing. I/O interfaces between the test tools and SUT are necessary for online MBT to realise a synchronised process of test generation and test execution. Therefore, online MBT is significantly more difficult to implement than offline MBT.

3.1.2 Online MBT for TCS

The advantages of offline testing can be summarised as follows:

- The test generation process can be controlled by the test selection criteria via test tools, which means that the tester can adjust the test generation strategy according to specific test requirements, such as some certain states having to be covered, or some important transitions having to be run through.
- Coverage of the test cases generated can be conveniently measured by the test tools,

which means the tester can decide when to stop the test generation process based on the coverage performance obtained. Furthermore, the test selection criteria and specification model can be adjusted to improve the coverage performance if the coverage of the test cases generated does not satisfy the test requirements.

- Test cases are separately generated and executed, which means that the two processes do not influence each other. This is important for testing SUTs which contain very strong time restrictions (at millisecond level), because the test generation time may be too long to obtain an output within the strict time constraint if the test is implemented in online mode [100, 101]. Therefore, for those SUTs, offline testing is more rational.

However, the disadvantages of offline MBT are also obvious:

- Offline MBT is not eligible to deal directly with nondeterministic SUTs. SUTs have to be modelled in a deterministic format in offline MBT, which decreases the testing efficiency and increases the modelling difficulties.
- Although test generation can be controlled by test selection criteria, coverage may not be achieved as expected because the specification model is too complex for the computer to analyse. The reason is explained in detail in Chapter 6.3.1.
- Since the test cases are generated separately in abstract format, the testing efficiency can be influenced by the translation process between the abstract and real I/O, which can be time-consuming for a large set of test cases.

By comparison, online MBT can remedy the limitations of offline MBT, as summarised

below:

- Online MBT can deal with nondeterministic SUTs according to a nondeterministic specification model. In online MBT, one input is generated based on the current states of the specification model; thereafter, the input generated is executed by the real SUT. Due to the nondeterministic characteristics, the expected output can be a set containing all acceptable output values, which is different from offline testing in which only one output corresponds to one input under the determined conditions.
- Online MBT is suitable for exhaustive testing of SUTs [102]. Benefiting from simultaneous test generation and execution, the SUT can be continuously tested for a relatively long time, depending on an appropriately built specification model which does not contain any deadlock and has reachability in all states. Although the test generation process cannot be guided by different test selection criteria, a decent level of coverage can still be achieved with online MBT because of the exhaustive feature, the reason for which is discussed in section 6.3.1.
- Without supervision of the test selection criteria, online MBT does not carry the risk of running out of memory because the test tools do not need to record any information to calculate coverage.

Along with the benefits brought by online MBT, the disadvantages cannot be ignored:

- For both online and offline MBT, an interface or adaptor is needed to map the abstract behaviour in the test generated and in the real data or command which is recognised

by the real SUT to execute it. Offline MBT translates the test cases generated into real data in offline mode, while online MBT needs to synchronise the translation process with the test generation process, which significantly increases the implementation difficulties.

- Strict time constraints may lead to a failed testing result in online MBT. Since the input is generated then executed in online MBT, the output result derived from execution of the input may not be able to be collected in time if the time constraints are very critical, because the processing capacity of the computer is limited. For example, if the specification model requires that an output should be observed 1 millisecond after the input is executed, the SUT may not pass the test because the input cannot be delivered to the SUT in time by a computer of average performance, even though in practice the output can be delivered in time by the SUT. Offline MBT does not suffer the same problem because the test cases are generated first and executed afterwards, which means the computer has sufficient time to generate the test cases, translate them into executable form, and directly execute them in the end.
- Without the guidance of the test selection criteria, the tester cannot directly judge the performance of test generation according to the coverage. The only way to find out the coverage performance is to analyse the testing log file after the test is finished, which can be time-consuming and inaccurate compared with offline MBT. In offline MBT, the coverage performance can be obtained after the test generation process and before the test execution process, which is a great advantage over online MBT when testing

SUTs for which the test execution processes take a relatively long time.

- To guarantee the testing efficiency for detecting errors, online MBT requires the tester to be more experienced in modelling and testing. As introduced in section 3.1.1, the testing process will be interrupted if an inconsistency is found between the SUT and the specification model, which means only one defect can be found in one implementation of online MBT. In offline MBT, a set of test cases are derived from the specification model and can be executed following a sequence. Assuming 10 errors contained in the SUT are evenly distributed into each test case, then offline MBT can locate all 10 errors by running the test process once, while online MBT needs to be run at least 10 times to locate all the errors. Even worse, online MBT can take far more time to run than offline MBT if the tester is not experienced enough to efficiently eliminate the errors located. Therefore, the tester's ability to eliminate errors in time is more important in online MBT than in offline MBT.

According to the advantages and disadvantages of offline and online MBT, the author chose online MBT as the MBT method. The reason is that the testing theme in this thesis mainly concentrates on the functional testing of TCSs implemented on a simulated network, which inevitably contains nondeterminism and has no strong time restrictions in the system specification [103]. The author has improved on existing MBT methods by introducing simulation combined MBT, which is explained in detail in the remaining sections of this chapter. To overcome coverage-related limitations, the methods of analysing the coverage performance of online testing are discussed in Chapter 6.

3.1.3 Introduction of Simulation Combined MBT

As a branch of MBT, the recent development of online MBT for the solution of MBT for nondeterministic SUTs has been rapid. Different modelling methods and corresponding different test tools are used to implement online MBT in different fields of SUTs. No matter what kind of modelling method or test generation tool is adopted, online MBT faces an unavoidable problem which is equally challenging for other types of MBT: formally modelling the SUT behaviour according to its specification requirements. The functional specification requirements of a system describe its behaviour by specifying a series of system functions in a series of operational scenarios, which means that those system functions can only be realised or valid when the system is operating in the corresponding scenarios [104]. Therefore, the tester needs to take the operational scenarios into consideration when building the specification model for implementation of MBT. The specification model is also known as the model of Implementation Under Test (IUT), in which implementation means the integrated system behaviour combining the SUT and its operational environment [105]. As a result, an IUT model can be divided into two main components, the SUT model and the environment model, as depicted in the left-hand part of Fig 13. In this modelling structure, the SUT behaviour is formally described as interactions between the SUT and its operational environment. The benefit of the IUT modelling structure is that it agrees with the normal form of black-box testing where inputs are generated out of the black box (the SUT) and outputs are delivered to the external observer (the environment), which is convenient for the tester to build the specification model. However, the modelling structure expands the size of the

specification model when the SUT specification requirements contain complex functions and a vast amount of different operational scenarios. To avoid high processing loads on the computer, as well as to decrease modelling difficulties, the author introduces simulation combined MBT, which is an evolution of traditional online MBT methods.

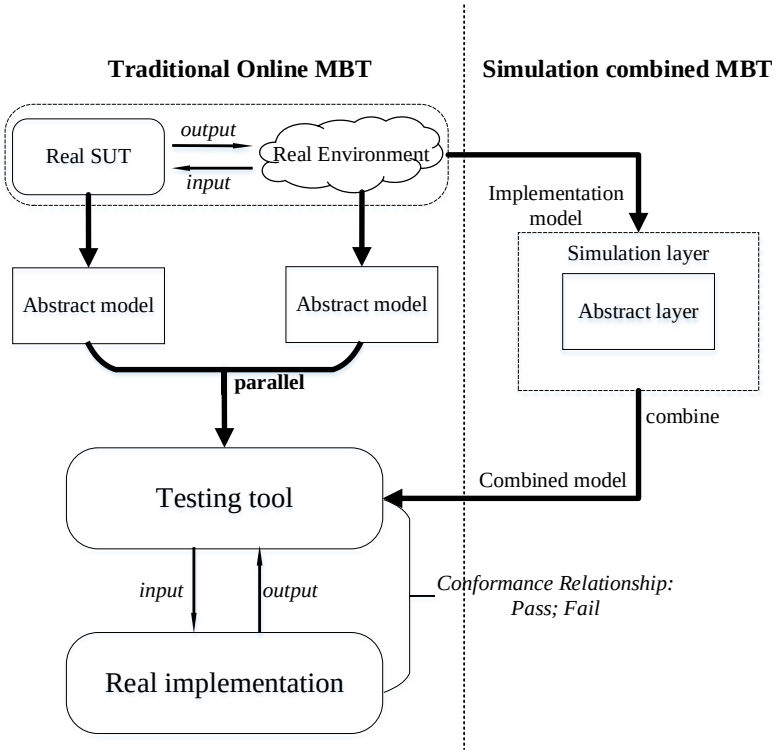


Fig 13 Structure of traditional MBT and simulation combined MBT

As shown in Fig 13, the differences between traditional MBT and simulation combined MBT are that they use different modelling methods to build the IUT model for the test tool. Traditional MBT models the SUT and its operational environment using the same formal method and uses the test tool to analyse the parallels of the models developed. Different from traditional MBT, the author proposes simulation combined MBT, which models the SUT and its operational environment in two models, the abstract model and the simulation model. As

shown by Fig 13, simulation combined MBT firstly combines the real SUT and environment into the real IUT, then build the IUT model in two models, where the abstract model is designed to describe the discrete and abstract behaviours of the IUT, and the simulation model is designed to deal with continuous variables and relatively complex calculations. Based on the combined model obtained, the test tool generates valid inputs and executes them simultaneously as it does in traditional online MBT. The outputs received are compared with the expected ones specified by the IUT model, to determine whether there is inconsistency. If no inconsistency is found, the conformance relation between the SUT and the implementation model is satisfied, and the test will end with a '***Passed***' result [106]. If any inconsistency is found, the test will end with a '***Failed***' result.

Compared with traditional MBT, simulation combined MBT significantly reduces the size of the formal implementation model by moving some non-vital elements into the simulation model. As a result, the test tool can test a more complicated SUT with the assistance of simulation without the risk of state explosion. Since the kernel function of the SUT is still modelled by the formal modelling method, the accuracy of the testing results will not be influenced by a two-model-combined modelling structure. However, the evolved structure of simulation combined MBT means that existing MBT architecture is no longer feasible, and a new configuration of the elements in MBT must be developed to adapt the two-model-combined structure. As indicated by Fig 13, three elements are essential for realising MBT, the modelling method used to build the specification model, the conformance relation for determining whether the SUT complies with the IUT model, and the test tool for

test execution. These key elements are explained in detail in Chapters 3 and 4.

3.2 Simulation Combined MBT

As introduced in section 3.1.3, simulation combined MBT is an evaluation of from the traditional online MBT methods which have been developed by previous research, such as those introduced by Larsen et al. [107] and Keranen and Raty [100]. Their research includes combining online MBT with simulation environment for embedded system testing. However, IUTs are modelled only by a formal method, while it is modelled by formal methods and simulation methods in simulation combined MBT. Some other commercial test tools support online MBT, which can be found in Table 4.

It should be noted that IUT in simulation combined MBT does not need to be modelled separately from the SUT and environment as it does in traditional MBT. In simulation combined MBT, IUT can be modelled in a combination of abstract and simulation models. With the two-model-combined structure, the tester can decide how to divide the SUT behaviour into the two models, which provides greater flexibility compared with traditional MBT. Especially for complex SUTs such as TCSs which contain intricate data-exchanging processes and state-transition flows; here, the advantages of the two-model-combined modelling structure can be better reflected because a single formal modelling method may not be able to individually accommodate all essential IUT factors. Unlike formal modelling methods, which model systems in abstract format, simulation builds system models from direct conversion of the system specification requirements, which is less difficult than formal

modelling because the complex conversions from natural language to formal expressions are unnecessary.

Simulation combined MBT is still a form of online MBT, and as such it inherits some similarities from traditional MBT methods, including the essential elements of the implementation of online MBT. As one of the most important basics of online MBT, modelling method has a huge influence on testing implementation because the other elements of online MBT such as test selection criteria and test tools are all determined by the modelling method. Therefore, in the following sections of Chapter 3, the author focuses on introducing the modelling method with a series of formal definitions, including the formal definition of the conformance relation adopted. Eventually, the modelling method of simulation combined MBT is formally defined so that the modelling feasibility can be preliminarily proven in theory.

3.2.1 Modelling for Online MBT

As mentioned previously, simulation combined MBT models the IUT using an abstract model and a simulation model. The simulation model can be written by mainstream programming languages, leaving only the formal modelling language of the abstract model to be determined. According to the discussion in section 2.2.1, different modelling methods are adept at depicting various characteristics of systems, so that selecting an appropriate modelling method is one of the key issues in MBT if model precision is to be guaranteed, because an inappropriately chosen modelling method can not only increase modelling difficulties but also

lead to a model deficient of essential SUT information. As explained in section 2.2.1.4, state-based modelling methods are more suitable for modelling data-oriented SUTs, while transition-based modelling methods are more suitable for modelling control-oriented methods. However, TCSs are highly integrated systems with a large number of different functions that are realised by both complex data-exchanging process and state-transition controlling flows, which means that TCSs can be both data-oriented and control-oriented systems. In accordance with the introduction which contains analysis of the system characteristics of ETCS and CBTC systems, the functions of all modern TCSs contain continuous variable manipulations and discrete state transitions. For example, the OBU continuously monitors the vehicle speed and sends out the emergency brake (EB) command once the vehicle speed is found to exceed the maximum speed limit, which is realised by continuously manipulating the variable ‘speed’ and making a state transition happen when the condition is satisfied. To test such a function, both variable manipulation and state transition should be taken into the consideration. Although the formal modelling method is mainly applied to model discrete IUT behaviour in simulation combined MBT, it still needs to be capable of reflecting key variable manipulations when important state transitions happen. As a result, the author has selected TA as the modelling theory, which supports modelling of IUT in FSM format with time constraints.

In the theory of TA, system behaviour can be described in the format of a Timed I/O Transition System (TIOTS) [43], which is an evolution of LTS by adding time constraints to states and transitions [108]. Definition 1-6 formally explain the modelling method of TIOTS

that is the modelling method for traditional MBT. In LTS, a system is divided into state nodes and transitions from node to node, where actions can happen when valid transitions are accessible. SUT behaviour is formally described in an LTS by profiling its static conditions with states and capturing its dynamic movements by actions on transitions. It should be noted that a transition can only happen when its conditions are satisfied. **Definition 1** presents a formal definition of an LTS which is developed by [109], with an example given by Fig 14 which is a schematic of an LTS with four states and three transitions:

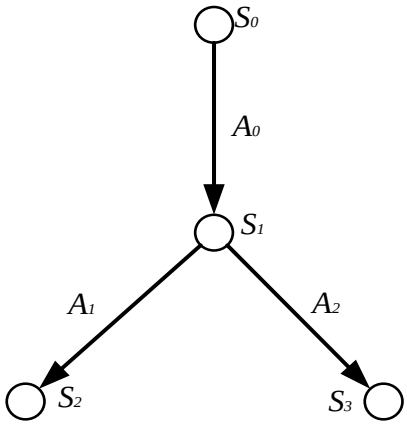


Fig 14 Schematic of an LTS

Definition 1: An LTS $\mathcal{A}_L$ is a quadruple tuple (S, So, A_τ, Tr), where

- S is a finite, non-empty set of states, where in Fig 14, $S = \{S_0, S_1, S_2, S_3\}$;
- So is the initial state, where in Fig 14, $So = S_0$;
- A_τ is a set of actions, including observable actions A and unobservable actions $\{\tau\}$, where $A_\tau = A \cup \{\tau\}$ and $\tau \notin A$ hold. It should be noted that the unobservable actions $\{\tau\}$ can be internal actions or silent actions indicating that $\mathcal{A}_L$ is in a quiescent state [110]. In Fig 14, $A = \{A_0, A_1, A_2\}$;

- Tr is set of transitions, where $Tr \subseteq S \times A\tau \times S$ holds. In Fig 14, $Tr = \{(S_0, A_0, S_1), (S_1, A_1, S_2), (S_1, A_2, S_3)\}$.

Therefore, the entire LTS given by Fig 14 and can be formally obtained by:

$$\mathcal{A}_L = (\{S_0, S_1, S_2, S_3\}, S_0, \{(S_0, A_0, S_1), (S_1, A_1, S_2), (S_1, A_2, S_3)\})$$

After the formal definition of the LTS is obtained, the state transition relations can be formally defined by **Definition 2**.

Definition 2: A trace is a sequence of observable actions derived from a transition sequence of an LTS.

Assuming the LTS $\mathcal{A}_L$ in **Definition 1** contains a transition sequence:

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \dots s_{k-1} \xrightarrow{a_{k-1}} s_k \quad (1)$$

where

$$\{s_0, s_1, \dots, s_{k-1}, s_k\} \subseteq S, s_0 \in So \quad (2)$$

$$\{a_0, a_1, \dots, a_{k-2}, a_{k-1}\} \subseteq A, where\ k \in \mathbb{N} \quad (3)$$

then a trace σ of the LTS $\mathcal{A}_L$ can be written as:

$$\sigma = a_0 \cdot a_1 \cdot \dots a_{k-1} \quad (4)$$

where $a_m \cdot a_n$ denotes the concatenation of a_m and a_n. Assuming all traces contained in $\mathcal{A}_L$ are A^*, then $\sigma \subseteq A^*$. When $s \in S, s' \in S$ and $a_i \in A_\tau$, the following definition holds:

According to (1) and (4), then

$$s \overset{\sigma}{\rightarrow} s' \overset{def}{=} \exists\{s_m, \dots, s_n\}: s_m \overset{a_m}{\rightarrow} \dots \overset{a_n}{\rightarrow} s_n ,$$

$$where\ s = s_m, s' = s_n, \sigma = (a_m \cdot \dots \cdot a_n), m, n \in \mathbb{N}\ and\ m < n \quad (5)$$

According to (5), then

$$s \overset{\sigma}{\rightarrow} \overset{def}{=} \exists s': s \overset{\sigma}{\rightarrow} s' \quad (6)$$

$$s \overset{\sigma}{\nrightarrow} s' \overset{def}{=} \nexists s': s \overset{\sigma}{\rightarrow} s' \quad (7)$$

In black-box testing, only external actions are observed, and internal actions are isolated from the view of the tester or test tool, which means that the tester can only observe quiescent IUT behaviour when internal actions or silent actions happen. Considering the remaining flexibility of test implementation, quiescent behaviour should be acceptable for testing so that the quiescent behaviour should be formally described in the LTS [110]. Therefore, the empty trace $\mathcal{E}$ is introduced to formally define the quiescent behaviour of IUT in LTS.

$$s \overset{\mathcal{E}}{\Rightarrow} s' \overset{def}{=} s = s'\ or\ s \overset{\sigma_\tau}{\rightarrow} s', where\ \sigma_\tau = (\tau \cdot \dots \cdot \tau) \quad (8)$$

$$s \overset{a_i}{\Rightarrow} s' \overset{def}{=} \exists\{s_m, s_n\}: s \overset{\mathcal{E}}{\Rightarrow} s_m \overset{a_i}{\rightarrow} s_n \overset{\mathcal{E}}{\Rightarrow} s', where\ m, n, i \in \mathbb{N}\ and\ m < n \quad (9)$$

According to (8) and (9), (5) and (6) can be redefined by including quiescent behaviour:

$$s \overset{\sigma}{\Rightarrow} s' \overset{def}{=} \exists\{s_m, s_n\}: s_m \overset{a_{m+1}}{\Longrightarrow} s_{m+1} \overset{a_{m+2}}{\Longrightarrow} \dots \overset{a_n}{\Rightarrow} s_n, where\ m, n \in \mathbb{N}\ and\ m + 2 \leq n \quad (10)$$

$$s \overset{\sigma}{\Rightarrow} \overset{def}{=} \exists s': s \overset{\sigma}{\Rightarrow} s' \quad (11)$$

According to (10) and (11), then

$$Trace(s) \overset{def}{=} \left\{\sigma \in A^* | \, s \overset{\sigma}{\Rightarrow}\right\} \tag{12}$$

$$s \; \boldsymbol{AFTER} \; \sigma \overset{def}{=} \{s' | s' \in S, s \overset{\sigma}{\Rightarrow} s'\} \tag{13}$$

Therefore, the set of all observable traces starting from state s can be defined by (12), and the reachable state s' after a trace σ which starts from state s can be defined by (13). According to the LTS given by Fig 14, the following equations can be obtained:

$$Trace(S_0) = \{\mathcal{E}, A_0, A_1, A_2, (A_0 \cdot A_1), (A_0 \cdot A_2)\}$$

$$S_0 \; \boldsymbol{AFTER} \; A_0 = \{S_1\}, S_1 \; \boldsymbol{AFTER} \; A_1 = \{S_2\}, S_1 \; \boldsymbol{AFTER} \; A_2 = \{S_3\}$$

After defining the relations between trace, state and action, IUT behaviour can be formally expressed in a format understandable by computers. However, to implement black-box testing in the MBT frame, the input actions and output actions need to be distinguished in the IUT model, indicating the communication process of I/O actions between the SUT and the environment. Therefore, the IOTS is introduced by [111] to refine the actions in LTS into I/O actions.

Definition 3: An IOTS $\mathcal{A}_{IO}$ is an LTS (S, So, $A\tau$, Tr) where I/O actions are disjointed for testing purposes. Assuming the input actions A_I and the output actions A_O are contained in A where $A\tau = A \cup \{\tau\}$, $s, s' \in S$ holds, then:

$$A = A_I \cup A_O, A_I \cap A_O = \Phi$$

For an IOTS, inputs are enabled in any state [105, 112]; from (10) and (11), then

$$\boldsymbol{whenever}\ s \overset{\sigma}{\Rightarrow} s'\ \boldsymbol{then}\ \forall a \in A_I : s' \overset{a}{\Rightarrow} \tag{14}$$

According to (14), it is indicated that all inputs can be enabled through internal transitions or external transitions in IOTS, which is called weak input enabling [104, 110, 113]. In contrast, inputs can only be enabled via external transitions in I/O automata, which is called strong input enabling [114]. Input enabling requires that a system should never refuse an input when it is delivered. It should be noted that unobservable actions τ in LTS change their meaning in IOTS. In the IOTS frame, a trace ending with actions τ indicates that output actions are absent in the corresponding states, which becomes an observable event. It can happen when an output is refused after an input is delivered. In black-box testing, output refusal is sometimes expected to be observed in the test sequence, and other inputs need to follow that event according to the specification requirements. Therefore, the traces containing quiescent transitions should be formally defined:

$$s \overset{\delta}{\rightarrow} s \overset{def}{=} \forall a \in A_O \cup \{\tau\} : s \overset{a}{\nrightarrow}, \boldsymbol{where}\ \delta \notin A \tag{15}$$

where the action δ denotes the observable event of output absence. From (15), then (12) can be extended into the IOTS to describe observable traces.

$$Trace(s) \overset{def}{=} \left\{\sigma \in (A \cup \delta)^* \mid s \overset{\sigma}{\Rightarrow}\right\} \tag{16}$$

Therefore, Trace(s) includes the δ transitions defined in (15) so that the formal descriptions of a trace with I/O actions and δ actions are obtained in the IOTS. The

expression of s **AFTER** σ stays the same, where $\overset{\sigma}{\Rightarrow}$ now includes the $s \overset{\delta}{\rightarrow} s$ defined in (15).

By extending the LTS to the IOTS, IUT behaviour can be formally described with differing I/O actions, which are two of the main objects inspected in black-box testing. However, testing a timed system requires that the system behaviour under time constraints should be formally depicted. To achieve this, the concept of a timed I/O transition system (TIOTS) [107] is introduced to obtain an IUT model in the real-time region. A schematic of a TIOTS is given in Fig 15:

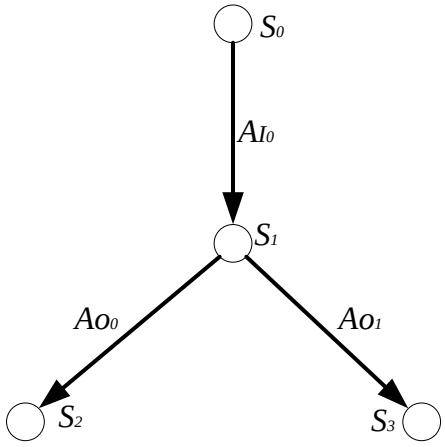


Fig 15 Schematic of a TIOTS

Definition 4: A TIOTS $\mathcal{A}_T$ is a quintuple (S, S_O, A_I, A_O, Tr^T), where

- S is a finite, non-empty set of states, where in Fig 15, $S = \{S_0, S_1, S_2, S_3\}$;
- S_O is the initial state, where in Fig 14, $S_O = S_0$;
- A_I and A_o denote the observable I/O actions which have been defined in the IOTS. It should be noted that the action δ mentioned in the IOTS is extended to delay actions in the TIOTS because quiescent actions can be reflected as time delays in the time

region. $A_{\tau\delta}$ is a set of actions containing observable actions $A = A_I \cup A_O$, unobservable actions $\{\tau\}$ and observable delay actions $\{\delta\}$, where

$$A_{\tau\delta} = A \cup \{\tau\} \cup \{\delta | \delta \in \mathbb{R} \geq 0\},\ \tau \notin A,\ \delta \notin A;$$

$$A_\tau = A \cup \{\tau\},\ A_\delta = A \cup \{\delta\}$$

In Fig 15, $A = \{A_{I_0}, A_{O_0}, A_{O_1}\}$;

- Tr^T is a set of transitions, where $Tr \subseteq S \times A_{\tau\delta} \times S$ holds, presenting a set of transition relations under a set of time constraints. Therefore, the state transitions observed in the time region can be written as state sequences with time intervals $s_0 \xrightarrow{\delta_0} s_1 \xrightarrow{\delta_1} \ldots s_n \xrightarrow{\delta_n} \ldots$. which should then satisfy the following properties:
    - Time determinism:

$$\forall s \in S, \exists s', s'' \in S: \left(s \xrightarrow{\delta} s'\right) \wedge \left(s \xrightarrow{\delta} s''\right)\ iff\ s' = s'' \qquad (17)$$

    - Time additivity:

$$\forall s, s' \in S, \exists s'' \in S, \delta_1, \delta_2 \in \delta: s \xrightarrow{\delta_1} s'' \xrightarrow{\delta_2} s'\ iff\ s \xrightarrow{\delta_1+\delta_2} s' \qquad (18)$$

    - Zero delay:

$$\forall s, s' \in S: s \xrightarrow{0} s'\ then\ s = s' \qquad (19)$$

Based on **Definitions 1**, **2** and **3**, the I/O actions observable with observable time intervals can be formally defined as follows:

Letting $a, a_0, a_1, \ldots, a_n \in A, and\ \alpha, \alpha_0, \alpha_1, \ldots, \alpha_n \in A_{\tau\delta}, and\ \delta, \delta_0, \delta_1, \ldots, \delta_n \in \mathbb{R} \geq 0$, then according to (5) and (6), a transition sequence $\mathcal{T}$ of the TIOTS $\mathcal{A}_T$ can be obtained:

$$\mathcal{T} = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \dots \xrightarrow{\alpha_{n-1}} s_n \quad (20)$$

As indicated by (20), the transition starts from s_0 and ends in s_n, where $s_n \overset{\alpha}{\nrightarrow}$ holds so that s_n is the destination state for the transition sequence in (20). Since $\alpha, \alpha_0, \alpha_1, \dots, \alpha_n \in A_{\tau\delta}$, $\mathcal{T}$ can be decomposed into the I/O action-enabled transitions $s \overset{a}{\Rightarrow} s'$ and the delay-enabled transitions $s \overset{\delta}{\Rightarrow} s'$, which are defined by (21) and (22):

$$s \overset{a}{\Rightarrow} s' \overset{def}{=} \exists\{s_m, s_n\}: s \overset{\tau}{\Rightarrow} s_m \xrightarrow{a} s_n \overset{\tau}{\Rightarrow} s', where\ m, n \in \mathbb{N}\ and\ m < n \quad (21)$$

$$s \overset{\delta}{\Rightarrow} s' \overset{def}{=} s \overset{\tau}{\Rightarrow} s_1 \xrightarrow{\delta_1} s_2 \overset{\tau}{\Rightarrow} \dots \overset{\tau}{\Rightarrow} s_{m-1} \xrightarrow{\delta_n} s_m \overset{\tau}{\Rightarrow} s',$$

$$where\ m, n \in \mathbb{N}\ and\ m = 2n, \delta = \textstyle\sum_{i=1}^{n} \delta_i \quad (22)$$

Based on (21) and (22), $s \overset{a}{\Rightarrow}$ and $s \overset{\delta}{\Rightarrow}$ are written to represent all transitions starting from state s. To model IUT with TIOTS, it is necessary to define the following properties:

- Weak input enabling

  As defined in (14), input enabling systems cannot refuse input action. Strong input enabling can only enable input actions via external transitions, while weak input enabling can enable input actions via both external and internal transitions, which is proven below.

  Letting $\mathcal{P}_{WIE}(\mathcal{A}_T)$ be the property of weak input enabling of $\mathcal{A}_T$, and $\mathcal{P}_{SIE}(\mathcal{A}_T)$ be the property of strong input enabling of $\mathcal{A}_T$, which are defined by (23) and (24):

  $$\mathcal{P}_{SIE}(\mathcal{A}_T)\ iff\ \forall s \in S, a_i \in A_I: s \xrightarrow{a_i} \quad (23)$$

$$\mathcal{P}_{WIE}(\mathcal{A}_T)\ iff\ \forall s \in S, a_i \in A_I : s \overset{a_i}{\Rightarrow} \qquad (24)$$

- Non-blocking

  Letting $\mathcal{P}_{NB}(\mathcal{A}_T)$ be the non-blocking property of $\mathcal{A}_T$, then

$$\mathcal{P}_{NB}(\mathcal{A}_T)\ iff\ \forall s \in S, a_{o_i} \in A_O, t \in \mathbb{R} \geq 0 : s \overset{\sigma}{\Rightarrow},$$
$$where\ \sigma = \delta_0 \cdot a_{o_0} \cdot \delta_1 \cdot a_{o_1} \cdot \ldots \cdot \delta_n \cdot a_{o_n}, \sum_{i=1}^{n} \delta_i \geq t \qquad (25)$$

  According to (25), the time of the TIOTS $\mathcal{A}_T$ is not blocked by the environment when its successor set of states is reachable after execution of a trace σ within an existing set of delays. Therefore, the TIOTS $\mathcal{A}_T$ does not block the time process in any enabled environment, and the time process of $\mathcal{A}_T$ cannot be influenced by the external environment either. This means that the TIOTS cannot urge input delivery from the environment, and the environment cannot force output generation from the TIOTS. As a result, the non-blocking property guarantees that the time passes equivalently in the TIOTS and the environment, neither being able to be interrupted by the other.

- Output determinism

  Output determinism is an important property of the TIOTS to avoid obtaining ambiguous output results. Otherwise, one input corresponding with more than one outputs will make it difficult for the judgement logic of the test tool to determine which output is the correct one.

Letting $\mathcal{P}_{\mathrm{OD}}(\mathcal{A}_{\mathrm{T}})$ be the output determinism property of the TIOTS $\mathcal{A}_{\mathrm{T}}$, then

$$\mathcal{P}_{OD}(\mathcal{A}_T) \overset{\boldsymbol{def}}{=} whenever\ \exists \alpha', s, s' \in S: s \overset{\alpha}{\rightarrow} s' \wedge s \overset{\alpha'}{\rightarrow} s'\ then\ \alpha = \alpha' \quad (26)$$

According to (26), the successor state of an executed action is always deterministic, and only one state corresponds to one action.

- Output isolation

Another important property related to output actions is output isolation, which requires that the TIOTS should only deliver one output at a time and should never withdraw the output delivered by executing unobservable internal actions or observable delay actions.

Letting $\mathcal{P}_{OI}(\mathcal{A}_T)$ be the output isolation property of the TIOTS $\mathcal{A}_T$, then

$$\mathcal{P}_{OI}(\mathcal{A}_T) \overset{\boldsymbol{def}}{=} whenever\ s \overset{a_o}{\Rightarrow}\ where\ s \in S\ and\ a_o \in A_O$$

$$then\ s \overset{\tau}{\not\Rightarrow} \wedge s \overset{\delta}{\not\Rightarrow} \quad (27)$$

Therefore,

$$whenever\ \exists a_o \in A_O, s, s' \in S: s \overset{a_o}{\Rightarrow} s' \wedge s \overset{a_o'}{\Rightarrow} s'\ then\ a_o = a_o' \quad (28)$$

- Output urgency

The last property is output urgency, which requires that the TIOTS should deliver the output immediately the output is ready, which means that delays do not exist between

the TIOTS and its environment.

Letting $\mathcal{P}_{OC}(\mathcal{A}_T)$ be the output urgency property of the TIOTS $\mathcal{A}_T$, then

$$\mathcal{P}_{OC}(\mathcal{A}_T) \overset{def}{=} whenever\ s \overset{a_o}{\Rightarrow} \vee\ s \overset{\tau}{\Rightarrow}\ where\ a_o \in A_O, s \in S$$

$$then\ \forall \delta \in \mathbb{R} > 0: s \overset{\delta}{\nRightarrow} \tag{29}$$

It should be noted that when modelling real IUT for the implementation of simulation combined MBT, the communication delay between the SUT and its operational environment should be included in the IUT model. The method of implementing modelling of communication delays is explained in section 5.1.1.

Based on the defined TIOTS and the necessary properties for black-box testing, the relations between observable actions, delays and state transitions can be formally defined.

Definition 5: A sequence of observable actions during implementation of black-box testing is an observable timed trace $\sigma \in A_\delta^*$, where * denotes abstract transition relations where transitions can be triggered by $\alpha \in A_\delta$. As defined in **Definition 4**, $A = A_I \cup A_O$ denotes all observable I/O actions contained in the TIOTS, and $\delta \in \mathbb{R} \geq 0$ represents time delays between I/O actions. Therefore, the observable timed trace σ is defined below:

$$\sigma = a_0 \cdot \delta_0 \cdot a_1 \cdot \delta_1 \cdot \ldots \cdot a_n \cdot \delta_n, where\ n \in \mathbb{N}_+ \tag{30}$$

According to (30), all timed observable traces $Tr^T(s)$ starting from state s can be obtained:

$$Tr^T(s) \overset{def}{=} \exists \sigma \in A_\delta^*: s \overset{\sigma}{\Rightarrow} \tag{31}$$

Therefore, for the state s and trace σ, if there exists a reachable state after execution of σ in state s, which is written as $s\ \boldsymbol{AFTER}\ \sigma$, then

$$s\ \boldsymbol{AFTER}\ \sigma \overset{\boldsymbol{def}}{=} \exists s' \in S : s \overset{\sigma}{\Rightarrow} s' \tag{32}$$

Hence, for the super set S' of the state s, where $S' \subseteq S$, the set of states reachable after execution of σ in state S' can be obtained:

$$S'\boldsymbol{AFTER}\ \sigma = \bigcup_{s \in S'} (s\ \boldsymbol{AFTER}\ \sigma) \tag{33}$$

According to (31) and (32), the observable input action and observable output action with delays which are derived from state s can be obtained:

$$IN(s) = \{a_i \in A_I | s \overset{a_i}{\Rightarrow}\}, OUT(s) = \{a_o \in A_O \cup \delta | s \overset{a_o}{\Rightarrow}\} \tag{34}$$

According to (33) and (34), the set of observable input actions with delays derived from the set of states S', where inputs are enabled, can be obtained:

$$IN(S') = \bigcup_{s \in S'} IN(s)\,, where\ S' \subseteq S \tag{35}$$

According to (33) and (34), the set of observable output actions with delays derived from the set of states S', where outputs are enabled, can be obtained:

$$OUT(S'') = \bigcup_{s \in S''} OUT(s)\,, where\ S'' \subseteq S \tag{36}$$

Using **Definition 5**, the elements which are necessary for modelling system behaviour based on TIOTS are formally defined. With expression of the relations between the actions which

can be observed during the functional black-box testing procedure and the state transitions, the essential information is provided for test tools to generate I/O sequences from the IUT model. One of the main tasks of online MBT is to maximally substitute manual functional black-box testing by automating the test case generation and execution processes. Therefore, it still adopts the classic implementation architecture of functional black-box testing in which the tester observes the externally observable actions happening between IUT and its operational environment, which means both IUT behaviour and its environment behaviour should be modelled in the TIOTS format. With **Definition 5**, single-system behaviour in TIOTS has been defined. To obtain a specification model suitable for black-box testing, parallel composition of the IUT and environment should be defined so that their interactions can be modelled in TIOTS format. Fig 16 depicts the two parallel TIOTSs of the IUT and its operational environment.

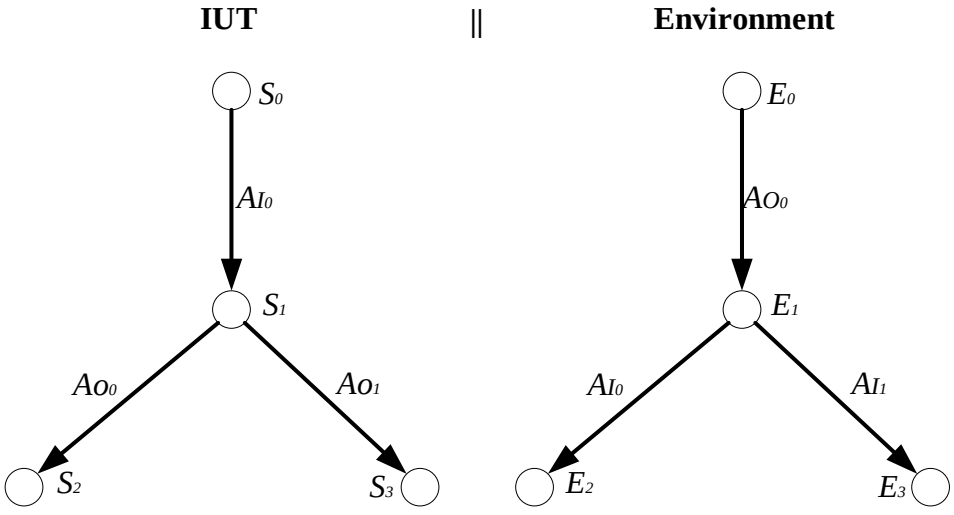


Fig 16 Schematic of the parallel configuration of two TIOTSs

Definition 6: Implementation of functional black-box testing I_B is a closed system where

IUT $\mathcal{S}$ and its operational environment $\mathcal{E}$ interact with each other in parallel.

$$I_B = \mathcal{S} \parallel \mathcal{E} \tag{37}$$

In previous definitions, I/O actions are defined from the perspective of the IUT, where input actions are delivered by the environment and output actions are sent out from the IUT. Therefore, from the perspective of the environment, a reversed form of I/O actions can be obtained. According to **Definition 4**, the IUT and its operational environment can be formally expressed as two TIOTSs:

$$\mathcal{S} = (S, S_0, A_I, A_O, Tr^T), \mathcal{E} = (E, E_0, A_O, A_I, Tr^T) \tag{38}$$

In (38), S and S_0, respectively, denote all the states and the initial state of the IUT, and E and E_0, respectively, denote all the states and the initial state of the operational environment of the IUT. According to (37) and (38), the following properties can be satisfied, where $s \in S, e \in E,$ and $a \in A_I \cup A_O$.

$$whenever\ s \xrightarrow{a} s' \wedge e \xrightarrow{a} e'\ then\ (s,e) \xrightarrow{a} (s',e') \tag{39}$$

$$whenever\ s \xrightarrow{\tau} s'\ then\ (s,e) \xrightarrow{\tau} (s',e) \tag{40}$$

$$whenever\ e \xrightarrow{\tau} e'\ then\ (s,e) \xrightarrow{\tau} (s,e') \tag{41}$$

$$whenever\ s \xrightarrow{\delta} s' \wedge e \xrightarrow{\delta} e'\ then\ (s,e) \xrightarrow{\delta} (s',e') \tag{42}$$

Therefore, the behaviour of the system in the parallel TIOTS is formally defined. It should be noted that in the real implementation of online testing of a TCS, some further information is

required for observable actions, including variables along with related manipulations in states or transitions, guards controlling whether transitions are accessible based on the current values of variables, and clocks [115]. However, the fundamental framework of the formal model should abide by the defined composition based on TA theory, and the remaining information can be defined by specific modelling tools. To cope with the defined modelling format, the tester should focus on observable I/O actions with relevant time constraints and omit internal actions which are unobservable, which is similar to the implementation structure of black-box testing.

3.2.2 Conformance relation in MBT [107]

According to Fig 13, the objective of MBT is to determine whether the real behaviour of the SUT complies with that described by the IUT model, which needs to be determined by computer. Therefore, a criterion needs to be formally defined to provide a formal standard for the test tools, determining compliance between the formal model and the real SUT [106]. The formal criterion is called the conformance relation which is adopted to judge whether the SUT behaviour complies with the specification requirements of the IUT model [116]. As shown in Fig 17, to prove that the IUT conforms to the specification requirements, the IUT behaviour should be proven to be a subset of IUT model behaviour, which means that different manners of describing the IUT behaviour lead to different conformance relations. The trace defined in **Definition 2** is adopted as the manner of describing the IUT behaviour for online MBT, so that the corresponding conformance relations are trace conformance relations [113]. To determine that the real IUT is trace conformant with the IUT model, various conformance

relations have been well developed for LTS-based models with different emphases. One relation, called the trace preorder relation, requires that IUT traces should not contain actions which are not included in the IUT model [117]. Another, stronger, relation requires that the IUT should not only perform the actions expected in the specification, but also refuse actions which cannot be performed in the IUT model [117]. The input–output conformance relation (*ioco*) requires that the IUT should produce an output only if it is one expected by the specification requirements which are presented by the IUT model [113]. Evolved from the *ioco*, symbolic *ioco* (*sioco*) is developed to define the conformance relations in a symbolic I/O system which is an extension of the IOTS [118].

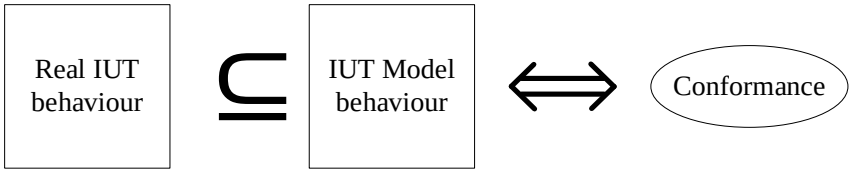


Fig 17 Schematic of the conformance relation in MBT

Since the IUT model is based on TA theory in this thesis, the *ioco* relation needs to be extended into the time region, which is called the timed *ioco* (*tioco*). The *tioco* requires that the IUT should never perform an action unexpected by the specification requirements or the IUT model, where unexpected actions include not only unexpected outputs but also violations of time constraints. The reason for adopting the *tioco* instead of another conformance relation is that it satisfies the requirements of black-box testing which concern external I/O actions [119]. For complex SUTs such as TCSs, the *tioco* relation inspects the crucial information provided by the specification requirements, such as the I/O sequence with time constraints,

and omits inspection of other non-vital information, to control the computational load during testing implementation. As a result, the *tioco* is adopted as the conformance relation for simulation combined MBT. The formal description of the *tioco* which extends the *ioco* into the time region is defined below:

$$i \ \boldsymbol{tioco} \ s \ iff \ \forall \sigma \in Tr^{T}(e).\, OUT((i,e) \ \boldsymbol{AFTER} \ \sigma) \subseteq OUT((s,e) \ \boldsymbol{AFTER} \ \sigma) \quad (43)$$

where $\boldsymbol{tioco}$ denotes: after executing any available timed trace σ based on the state $e \in E$ which is a state of the environment TIOTS, the set of destination states generated from the state i of the implementation TIOTS and the environment state e will always be a subset of the set of destination states generated from state s of the specification TIOTS and the environment state e [14]. '$\boldsymbol{AFTER}\ \sigma$' denotes all the reachable destination states achieved by the parallel systems after executing a timed I/O trace σ, which has been defined in (33). Whenever $i \ \boldsymbol{tioco} \ s$ is true, implementation i is determined to be conformant with specification s. It should be noted that the prerequisite of $\boldsymbol{tioco}$ is that both the IUT system i and the specification system s should perform behaviour under the constraint of the same operational environment e. Based on the formal definition of $\boldsymbol{tioco}$, consistency between the real IUT and the IUT model can be determined automatically by the computer during execution of online MBT.

3.2.3 Modelling method for Simulation Combined MBT

In sections 3.2.1 and 3.2.2, the modelling method and conformance relation of typical online MBT have been formally defined, which means that online MBT can be implemented if IUT

behaviour can be modelled in the manner defined. However, TCSs are highly complex, with numerous safety-related components and various operational scenarios. Modelling such systems with the methods defined can easily lead the formal models to a state explosion situation where the possibilities contained in the IUT model exceed the computational capability of the computer so that the testing fails to obtain valid testing results [120]. To apply MBT and to automate the functional testing of TCSs, the simulation combined online MBT method addresses the challenge of testing complex integrated systems and takes the advantages of both simulation and formal methods. As shown in Fig 13, the key part of simulation combined MBT is to model implementation by two models with two different modelling methods, rather than to model it by a single formal modelling method such as the one introduced in section 3.2.1 which realises the parallel structure S^A of the SUT and its environment defined by **Definition 6**. To realise automatic testing with the application of an online testing algorithm, the modelling method needs to be formally defined to adapt to the TIOTS format. According to **Definitions 1–6**, the refined modelling method for simulation combined MBT, which is an extension of the defined TIOTS, is formally defined by **Definition 7**. Considering the modelling method as dividing the system behaviour into a two-model-combined structure based on the current TIOTS frame, the author names it SCTIOTS.

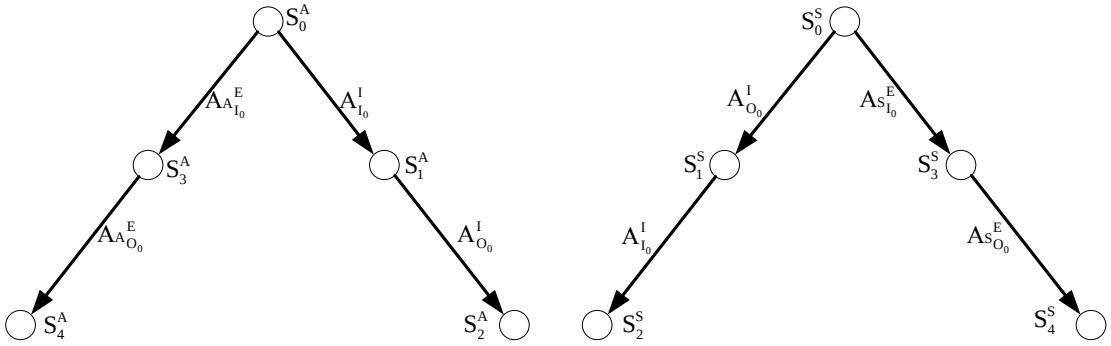


Fig 18 Schematic of an SCTIOTS

Definition 7: A Simulation Combined Timed I/O Transition System (SCTIOTS) is two TIOTSs in parallel, where $\mathcal{A}_{\mathrm{T}}^{\mathrm{S}} = \mathcal{S}_{\mathrm{A}} \parallel \mathcal{S}_{\mathrm{S}}$. $\mathcal{S}_{\mathrm{A}} = \left(S^A, S_O^A, A_I^I, A_O^I, A_{A_I}^E, A_{A_O}^E, Tr_A^T\right)$ is a septuple, modelling the implementation behaviour of the abstract model. $\mathcal{S}_{\mathrm{S}} = \left(S^S, S_O^S, A_O^I, A_I^I, A_{S_I}^E, A_{S_O}^E, Tr_S^T\right)$ is also a septuple, modelling the implementation behaviour of the simulation model.

In the TIOTS $\mathcal{S}_{\mathrm{A}}$:

- S^A is a finite, non-empty set of states, where in Fig 18, $S^A = \{S_0^A, S_1^A, S_2^A, S_3^A, S_4^A\}$;
- S_O^A is the initial state, where in Fig 14, $S_O^A = S_0^A$;
- A_I^I and A_O^I denote the observable internal I/O actions which are different from the internal actions defined in the TIOTS, where internal actions τ denote the internal unobservable actions in a TIOTS. The observable internal actions here refer to the I/O actions between the abstract model and the simulation model of the SCTIOTS. In contrast, $A_{A_I}^E$ and $A_{A_O}^E$ denote the observable external I/O which happen between the

$\mathcal{S}_{\mathrm{A}}$ and the environment. Similar to that in the TIOTS, $A_{\tau\delta}^{A}$ is a set of actions in the TIOTS $\mathcal{S}_{\mathrm{A}}$, containing observable actions $A^A = A^I \cup A_A^E$, unobservable actions $\{\tau\}$ and observable delay actions $\{\delta\}$, where:

$$A^I = A_I^I \cup A_O^I, A_A^E = {A_{A_I}}^E \cup {A_{A_O}}^E;$$

$$A_I^A = A_I^I \cup {A_{A_I}}^E, A_O^A = A_O^I \cup {A_{A_O}}^E;$$

$$A_{\tau\delta}^A = A^A \cup \{\tau\} \cup \{\delta | \delta \in \mathbb{R} \geq 0\},\ \tau \notin A^A,\ \delta \notin A^A;$$

$$A_\tau^A = A^A \cup \{\tau\},\ A_\delta^A = A^A \cup \{\delta\};$$

In Fig 18, $A_I^A = \{A_{I_0}^I, {A_{A_I}}^E{}_0\}$, $A_O^A = \{A_{O_0}^I, {A_{A_O}}^E{}_0\}$, $A^I = \{A_{I_0}^I, A_{O_0}^I\}$ and $A_A^E = \left\{{A_{A_I}}^E{}_0, {A_{A_O}}^E{}_0\right\}$.

- According to the defined TIOTS, Tr_A^T is a set of transitions in $\mathcal{S}_{\mathrm{A}}$, where $Tr_A^T \subseteq S^A \times A_{\tau\delta}^A \times S^A$ holds, and the properties of time determinism, time additivity and zero delay should all be satisfied. Moreover, the input is still weakly enabled, and the time cannot be blocked in the TIOTS $\mathcal{S}_{\mathrm{A}}$. Output properties such as output determinism, output isolation and output urgency still hold in $\mathcal{S}_{\mathrm{A}}$.

In the TIOTS $\mathcal{S}_{\mathrm{S}}$:

- S^S is a finite, non-empty set of states, where in Fig 18, $S^S = \{S_0^S, S_1^S, S_2^S, S_3^S, S_4^S\}$;
- S_O^S is the initial state, where in Fig 14, $S_O^S = S_0^S$;
- Similar to $\mathcal{S}_{\mathrm{A}}$, the observable internal I/O actions are A_O^I and A_I^I, which have a reversed order compared with $\mathcal{S}_{\mathrm{A}}$ to denote internal I/O actions in the parallel system of the two TIOTSs defined in **Definition 6**. Differently, ${A_{S_I}}^E$ and ${A_{S_O}}^E$ denote the observable external I/O which happen between $\mathcal{S}_{\mathrm{S}}$ and the environment. Similarly,

$A_{\tau\delta}^{S}$ is a set of actions in the TIOTS $\mathcal{S}_{\mathrm{S}}$, containing observable actions $A^{S} = A^{I} \cup A_{S}^{E}$, unobservable actions $\{\tau\}$ and observable delay actions $\{\delta\}$, where:

$$A^{I} = A_{O}^{I} \cup A_{I}^{I}, A_{S}^{E} = A_{S_I}^{E} \cup A_{S_O}^{E};$$

$$A_{I}^{S} = A_{I}^{I} \cup A_{S_I}^{E}, A_{O}^{S} = A_{O}^{I} \cup A_{S_O}^{E};$$

$$A_{\tau\delta}^{S} = A^{S} \cup \{\tau\} \cup \{\delta | \delta \in \mathbb{R} \geq 0\},\ \tau \notin A^{S},\ \delta \notin A^{S};$$

$$A_{\tau}^{S} = A^{S} \cup \{\tau\},\ A_{\delta}^{S} = A^{S} \cup \{\delta\};$$

In Fig 18, $A_{I}^{S} = \{A_{I_0}^{I}, A_{S_{I_0}}^{E}\}$, $A_{O}^{S} = \{A_{O_0}^{I}, A_{S_{O_0}}^{E}\}$, $A^{I} = \{A_{I_0}^{I}, A_{O_0}^{I}\}$ and $A_{S}^{E} = \left\{A_{S_{I_0}}^{E}, A_{S_{O_0}}^{E}\right\}$.

Note that $\mathcal{S}_{\mathrm{A}}$ and $\mathcal{S}_{\mathrm{S}}$ share corresponding (reversed) internal I/O actions but individually they have different external I/O actions interacting with the environment. Therefore, the following equations hold:

$$A_{I}^{A} \cap A_{I}^{S} = A_{I}^{I}, A_{O}^{A} \cap A_{O}^{S} = A_{O}^{I}, A_{A}^{E} \cap A_{S}^{E} = \Phi$$

- According to the defined TIOTS, Tr_{S}^{T} is a set of transitions in $\mathcal{S}_{\mathrm{S}}$, where $Tr_{S}^{T} \subseteq S^{S} \times A_{\tau\delta}^{S} \times S^{S}$ holds, and the properties time determinism, time additivity and zero delay should all be satisfied. Moreover, the input is still weakly enabled, and the time cannot be blocked in the TIOTS $\mathcal{S}_{\mathrm{A}}$. Output properties such as output determinism, output isolation and output urgency still hold in $\mathcal{S}_{\mathrm{S}}$.

Based on the defined SCTIOTS, IUT behaviour can be modelled in the form of two parallel TIOTSs. The TIOTS of the abstract model is utilised to describe the abstract behaviour of the IUT, and the TIOTS of the simulation model is utilised to model the concrete behaviour of the IUT. Only the abstract model is built by a formal modelling tool so that only the model of the

abstract model is analysed by the test tool in MBT implementation. As it has a connection with the abstract model, the IUT behaviour modelled in the simulation model can be tested indirectly. Since only the abstract model will be analysed by the test tool, the author has defined the relations between observable actions, delays and state transitions from the view of the abstract model of the SCTIOTS. Based on **Definition 5**, **Definition 8** formally defines the necessary elements for online MBT.

Definition 8: A sequence of observable actions during implementation of black-box testing is an observable timed trace $\Sigma \in \left(A_\delta^A\right)^*$, where * denotes abstract transition relations where transitions can be triggered by $a \in A_\delta^A$. From **Definition 7**, $A^A = A^I \cup A_A^E$ denotes all observable I/O actions contained in $\mathcal{S}_A$, including the observable internal I/O actions with $\mathcal{S}_S$ and the external I/O actions with the environment. $\delta \in \mathbb{R} \geq 0$ represents time delays observed in the transitions of $\mathcal{S}_A$. Letting $a, a_0, a_1, \ldots, a_n \in A^A, \alpha, \alpha_0, \alpha_1, \ldots, \alpha_n \in A_{\tau\delta}^A, and\ \Delta, \Delta_0, \Delta_1, \ldots, \Delta_n \in \mathbb{R} \geq 0$, the observable timed trace Σ is defined as:

$$\Sigma = a_0 \cdot \Delta_0 \cdot a_1 \cdot \Delta_1 \cdot \ldots \cdot a_n \cdot \Delta_n \qquad (44)$$

According to (44), all the timed observable traces $Tr_A^T(s)$ starting from state $s^A \in S^A$ can be obtained with:

$$Tr_A^T(s^A) \overset{\boldsymbol{def}}{=} \exists \Sigma \in \left(A_\delta^A\right)^* : s^A \overset{\Sigma}{\Rightarrow} \qquad (45)$$

Therefore, if for the state s^A and the trace Σ, there exists a reachable state after execution of Σ in state s^A, which is written as s^A ***AFTER*** Σ, then:

$$s^A \textbf{ AFTER } \Sigma \overset{def}{=} \exists s^{A'} \in S^A : s^A \overset{\Sigma}{\Rightarrow} s^{A'} \quad (46)$$

Hence, for the super set $S^{A'}$ of the state s^A, where $S^{A'} \subseteq S^A$, the reachable set of states after execution of Σ in state $S^{A'}$ can be obtained using:

$$S^{A'} \textbf{ AFTER } \Sigma = \bigcup_{s^A \in S^{A'}} (s^A \textbf{ AFTER } \Sigma) \quad (47)$$

According to (45) and (46), the internal input action and internal output action or the delay which are observable and derived from state s^A can be obtained:

$$IN^I(s^A) = \{a_i^I \in A_I^I | s^A \overset{a_i^I}{\Rightarrow}\}, OUT^I(s^A) = \{a_o^I \in A_O^I \cup \Delta | s^A \overset{a_O^I}{\Rightarrow}\} \quad (48)$$

According to (47) and (48), the internal input actions with delays derived from the set of states S_{II}^A where internal inputs are enabled can be obtained by:

$$IN^I(S_{II}^A) = \bigcup_{s^A \in S_{II}^A} IN^I(s^A), where\ S_{II}^A \subseteq S^A \quad (49)$$

According to (47) and (48), the internal output actions with delays derived from the set of states S_{IO}^A where internal outputs are enabled can be obtained by:

$$OUT^I(S_{IO}^A) = \bigcup_{s^A \in S_{IO}^A} OUT^I(s^A), where\ S_{IO}^A \subseteq S^A \quad (50)$$

Accordingly, the external input action and internal output action or delay which are observable and derived from state s^A can be obtained:

$$IN^E(s^A) = \{a_i^E \in A_I^E | s^A \overset{a_i^E}{\Rightarrow}\}, OUT^E(s^A) = \{a_o^E \in A_O^E \cup \Delta | s^A \overset{a_O^E}{\Rightarrow}\} \quad (51)$$

According to (47) and (51), the external input actions with delays derived from the set of

states S_{EI}^{A} where external inputs are enabled can be obtained by:

$$IN^{E}(S_{EI}^{A}) = \bigcup_{s^{A} \in S_{EI}^{A}} IN^{E}(s^{A}), where\ S_{EI}^{A} \subseteq S^{A} \quad (52)$$

According to (47) and (51), the external output actions with delays derived from the set of states S_{EO}^{A} where external outputs are enabled can be obtained by:

$$OUT^{E}(S_{EO}^{A}) = \bigcup_{s^{A} \in S_{EO}^{A}} OUT^{E}(s^{A}), where\ S_{EO}^{A} \subseteq S^{A} \quad (53)$$

According to (50) and (52), all observable input actions with delays of $\mathcal{S}_A$ can be obtained:

$$IN^{IE}(S_{IEI}^{A}) = IN^{I}(S_{II}^{A}) \cup IN^{E}(S_{EI}^{A}), where\ S_{IEI}^{A} = S_{II}^{A} \cup S_{EI}^{A} \subseteq S^{A} \quad (54)$$

According to (50) and (53), all observable output actions with delays of $\mathcal{S}_A$ can be obtained:

$$OUT^{IE}(S_{IEO}^{A}) = OUT^{I}(S_{IO}^{A}) \cup OUT^{E}(S_{EO}^{A}), where\ S_{IEO}^{A} = S_{IO}^{A} \cup S_{EO}^{A} \subseteq S^{A} \quad (55)$$

Based on **Definition 8**, the observable actions and delays are formally defined, which provides the possibility of automatic test generation via analysis of the defined formal model. The $\mathcal{S}_A$ of the SCTIOTS is still in the TIOTS architecture with classified internal I/O actions and external I/O actions. Therefore, the conformance relation between $\mathcal{S}_A$ and its operational environment still satisfies the one defined in section 3.2.2. Based on the refined definitions in **Definition 8**, the conformance relation between $\mathcal{S}_A$, which is the abstract model of the SCTIOTS $\mathcal{A}_T^S$, and a given environment can be rewritten:

$$i\ \boldsymbol{tioco}\ s\ iff\ \forall \Sigma \in Tr_A^T(e).\ OUT^{IE}((i,e)\ \boldsymbol{AFTER}\ \Sigma) \subseteq OUT^{IE}((s,e)\ \boldsymbol{AFTER}\ \Sigma) \quad (56)$$

where i represents implementation of the SCTIOTS $\mathcal{A}_T^S$ and the other elements stay as defined in the previous definitions. Based on the defined SCTIOTS consisting of abstract and simulation models, only the system behaviour in the abstract model $\mathcal{S}_A$ is formally modelled, which significantly reduces the formal model size. However, this modelling architecture cannot fully cover all the IUT behaviour included in the SCTIOTS model. The external I/O actions between the simulation model $\mathcal{S}_S$ and the environment are not inspected by the test tool because computational power may not be sufficient to cover inspection of all the I/O actions. However, the interactions proceed internally between the abstract model and the simulation model so that inconsistencies between the simulation model and the environment caused by external I/O actions can be indirectly detected by the test tool.

Based on the defined SCTIOTS and the corresponding conformance relation, the modelling framework of simulation combined MBT can be illustrated by Fig 19:

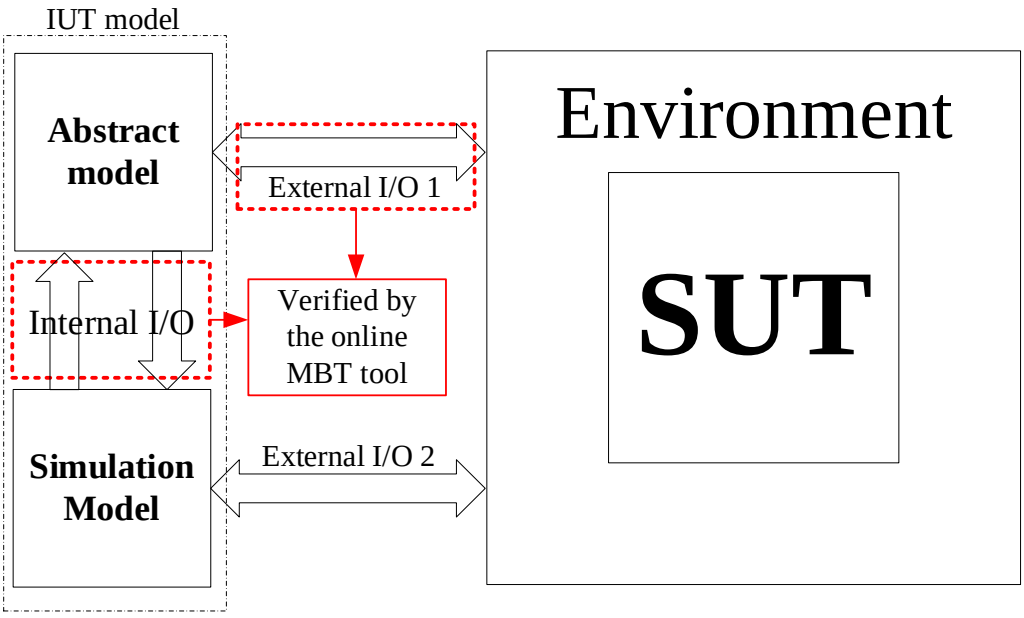


Fig 19 Modelling framework of simulation combined MBT

As revealed by Fig 19, simulation combined MBT is realised by a two-model-combined format of the IUT model. Meanwhile, I/O actions are divided into internal and external aspects as described in **Definition 8**. Different from the modelling method of typical MBT, which models individual components and obtains the implementation model by constructing parallel systems of the SUT and the environment, the SCTIOTS modelling method divides the original implementation into abstract and simulation models, where discrete and abstract actions are modelled by the abstract model, and continuous and specific variables are modelled in the simulation model. Based on the defined SCTIOTS, internal I/O actions between the abstract model and the simulation model can be formally described, and external I/O actions between the abstract model and the environment can be likewise described. According to the defined conformance relation, the I/O actions involving the abstract model can be automatically inspected by online MBT tools. As mentioned before, external I/O actions between the simulation model and the environment are exclusive of the inspections carried out by test tools but they can be indirectly and partially inspected via the internal I/O actions between the two models. Therefore, the consistency between the SUT and specification requirements can be determined. The remaining external I/O actions not covered are not verified during the test, but can be inspected according to the data recorded through the testing procedure.

With a testing purpose of black-box testing, the modelling method of MBT is required to be capable of modelling the entire SUT behaviour with unambiguous information, which can expand the model size to exhaust the computational power. The state space of the formal

model can easily exceed the memory of the computer when the formal model is too complex. Traditional modelling methods of online MBT have explicit modelling boundaries and architecture, such as the two parallel TIOTS systems introduced by **Definition 6**. The benefit of this modelling system structure is that the formal model can completely describe the expected behaviour of the SUT and its operational environment if the model size is acceptable for computation.

However, SUTs in industrial fields can be more complex than what can be afforded by computer. For example, an OBU should realise a series of functions involving other components or subsystems. Even for testing a single function, the OBU or its operational environment can be too complex for automatic test generation. To implement online MBT based on traditional modelling methods, both the SUT model and the environment need to be simplified to reduce the model size and complexity, which is time-consuming and carries a risk of losing information essential for testing. When applying SCTIOTS as the modelling method for online MBT, behaviour of the whole system can be modelled in two models, where a formal method builds the abstract model, and simulation builds the simulation model. With the SCTIOTS modelling framework, the formal modelling scale becomes adjustable so that that the tester can target the testing emphases of SUTs with limited computational power. This is especially significant for testing SUTs, such as TCSs, and other systems including numerous components with a complex structure.

The SCTIOTS modelling method introduces new problems which need to be solved. Firstly, the two-model-combined framework requires the tester to have a higher degree of modelling

skill, because the boundary between the abstract model and the simulation model is not a physical boundary such as that between the IUT and its operational environment. Instead, boundary is a logical boundary which can be flexibly defined by the tester, which means that the tester must be proficient in formal modelling and very familiar with the SUT operating principle, otherwise the use of poorly partitioned models can lead to poor testing accuracy and efficiency. Secondly, the SCTIOTS modelling method contains three kinds of I/O action while the traditional TIOTS modelling method only contains one; this increases the difficulty of building interfaces for the I/O channels of those I/O actions. Additionally, the priority of the I/O actions must be determined to avoid overwritten data or logical contradictions, which requires a further complex interface to appropriately synchronise the three kinds of I/O action. Finally, the uncovered functions modelled in the simulation model should be verified after online MBT is finished, which may take extra effort to configure the verification well according to the characteristics of the SUT.

3.3 Summary

In this chapter, the formal modelling method of simulation combined MBT is introduced as SCTIOTS which is an evolution of the existing TIOTS method. SCTIOTS differs from the TIOTS model, as it models the SUT into parallel abstract and simulation models, where the abstract model is developed using a formal modelling tool, and the simulation model is built by a simulation tool. The combination of formal modelling with simulation can significantly reduce the formal model size to avoid state explosion, which may exceed the computational power of the computer. Compared with TIOTS, SCTIOTS can be applied to model more

kinds of complex SUT because its application is not limited by the complexity of the SUT. Although SCTIOTS does require a more profound understanding of formal modelling and the SUT operating principle, it can be applied to more scenarios because of its flexibility, high degree of efficiency and interoperability.

4 Implementation of Simulation Combined MBT

4.1 Overview of the Simulation Combined MBT Platform

In Chapter 3, the modelling method for simulation combined MBT is explained, which provides the possibility of realising online MBT based on a simulation combined model. Evolved from traditional TIOTS, SCTIOTS significantly reduces the complexity of the specification model by dividing the IUT model into abstract and simulation models. However, the division approach poses new issues, which means that the original solutions of online MBT based on TIOTS cannot be transplanted to simulation combined MBT. The implementation of simulation combined MBT is explained in this chapter, which includes the steps in realising simulation combined MBT based on the theoretical method introduced. Fig 20 depicts a general solution of realising the simulation combined MBT introduced, which is named the simulation combined MBT platform.

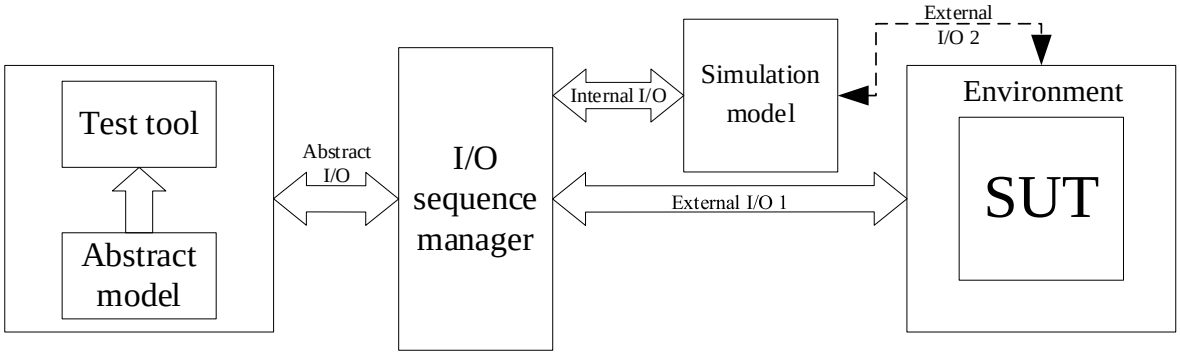


Fig 20 Architecture of the simulation combined MBT platform

As shown by Fig 20, the abstract model is built within the test tool where TRON takes charge of test generation based on the abstract model built by UPPAAL. The simulation model is

built by a microscopic railway simulator where the I/O behaviour can be simulated according to different SUTs. The I/O sequence manager is designed to manage the synchronisation relations between the internal I/O channel and external I/O channels. The environment is simulated to provide the HIL testing environment where SUT can operate as it would in the real operational environment. According to the architecture in Fig 20, simulation combined MBT automates system testing within an HIL environment, which means it is feasible to test a wide range of components in TCSs with minor modifications for different SUTs. Based on the HIL environment, various testing scenarios for different types of SUT can be implemented on the same testing platform, supporting automatic test generation and execution. The online feature of the testing platform makes it possible for testing to include more elements, without the risk of decreasing its accuracy or explosively expanding the model size. The testing platform can be smoothly run on a portable computer which can be conveniently brought to the testing field to test real hardware or software utilised in TCSs.

4.2 Modelling implementation of SUT

Since the model of simulation combined MBT is built in a combined model with abstract and simulation models, two modelling tools have been adopted to build the specification model. UPPAAL was adopted as the modelling tool for building the abstract model, and a microscopic railway simulator was adopted as the modelling tool for building the simulation model.

4.2.1 Modelling implementation of the Abstract Model

UPPAAL is a formal modelling and verification tool developed in collaboration between Uppsala University and Aalborg University. It supports modelling and verification of systems in real time, based on various types of formal model. Based on the theory of TA, UPPAAL perfectly supports the systems modelled in the TIOTS format. Since the abstract model of SCTIOTS is a variation of traditional TIOTS with division of internal and external I/O actions, UPPAAL is still suitable for building the abstract model in SCTIOTS format. Furthermore, UPPAAL integrates with a model-checking engine in the timed region, which is the basis of online test generation. Based on the known compatibility of UPPAAL with TIOTS, along with its well-developed toolboxes for system testing and verification, the author chose UPPAAL as the formal modelling tool to build the abstract model. The author will now present how to build an abstract model in UPPAAL by explaining elements contained by the model.

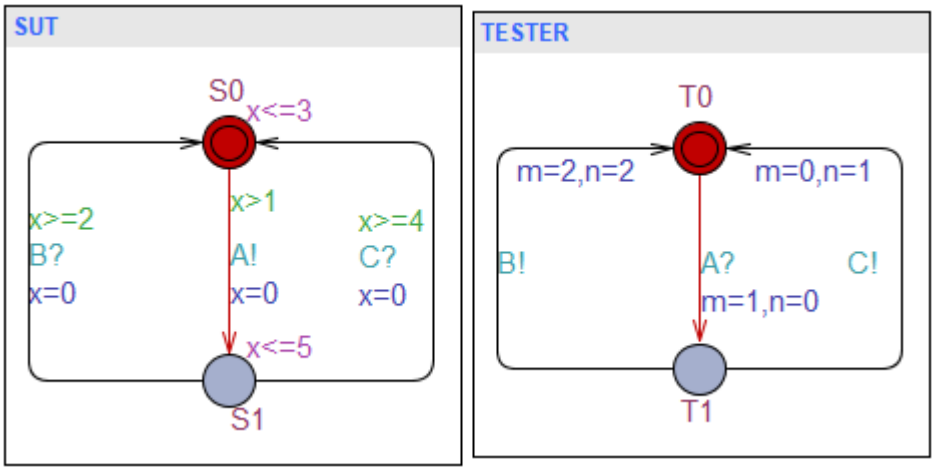


Fig 21 Example of the TA network model built in UPPAAL

Shown in Fig 21, a TA model network consists of one or more templates, the name of which is

given at the top left of each block, ‘SUT’ and ‘TESTER’, respectively in the case of Fig 21. In each template, the details of the system behaviour can be defined with a set of elements [121], which includes:

- **Locations** denote the states in the TIOTS. In locations, only the clock can accumulate, and other variables cannot be changed. Initial locations denote the initial states of the system, which in Fig 21 are the locations $\{S0, T0\}$. Urgent locations are a special location where time is not allowed to pass. Committed locations are more restrictive than urgent locations, where time is not allowed to pass, and the next execution of the system must include an outgoing edge if the system is in committed locations. Since urgent locations and committed locations are time-restrictive, the author tended to avoid using them when building the specification model because it can increase the computational load of the computer if too many urgent or committed locations are included.
- **Invariant** denotes the conditions that should be satisfied in locations. For example, the expression ‘x<=3’ in location ‘S0’ of the template SUT denotes that location ‘S0’ is only accessible when the time clock is no more than 3, which means that outgoing transitions must happen, and ingoing transitions must not happen when the time clock is more than 3. The author uses invariant to describe the time-out behaviour of the system where something must happen within a certain time.
- **Edges** denote the transitions contained in the system, which are always from one state to another. Time cannot accumulate on edges so that no time passes on edges. The transition on an edge can only be triggered when its guards are satisfied and when its synchronised

transition is also ready. Variables including the time clock can be updated after the transition on an edge is finished.

- **Synchronisation** is an expression denoting the synchronised behaviour between two or more templates, where two synchronised transitions must happen simultaneously with no order in succession. In Fig 21, ‘A!’ and ‘A?’ are a pair of synchronisations, which means the transition from ‘S0’ to ‘S1’ and the transition from ‘T0’ to ‘T1’ must happen together. If one of the transitions of the synchronised pair of transitions is not accessible, neither of the transitions in the synchronisation pair can happen. Synchronisation is a widely used expression in the specification model because it can denote the I/O channel where an output is sent from one component and received by another. In reality, however, the receiver side cannot receive the output immediately after it is sent out. Therefore, communication delays between each component should be taken into consideration when building the specification model.
- **Guard** is the condition that must be satisfied on a triggered transition. Guards can be used to build the selection structure, where the system chooses one of the valid actions to perform based on the value of the variable. In Fig 21, the expression ‘x>1’ on the edge ‘S0’ to ‘S1’ denotes that the transition can only happen when the time clock is more than 1.
- **Update** is the action that changes the variable values after a transition on an edge is finished. It is widely used in the specification model to denote a data value which is transmitted from one side to another or if the time clock of the system is reset by certain

actions. In Fig 21, the expression ‘x=0’ on the edge ‘S0’ to ‘S1’ denotes that the time clock will be reset to ‘0’ after the transition is finished and the system arrives at location ‘S1’, which means that the time clock accumulates from zero at location ‘S1’.

- **Select** is an expression that is adopted to denote nondeterministic values of a variable. By indicating the variable’s name and its accessible range of values, the system randomly updates a valid value when the corresponding transition happens. The author uses the select expression widely to describe the nondeterministic situations which can be observed during the testing procedure. The expression is explained in detail in section 5.1.1.

It should be noted that the model built by UPPAAL is a static model, which means that it cannot generate the inputs and outputs depicted in Fig 20. To generate the required inputs and outputs, the model established in UPPAAL needs to be analysed by the test tool TRON, which is explained in section 4.3. Based on UPPAAL with the expressions introduced, the system behaviour is formally described in the TIOTS format, which provides a specification for the test tool TRON to generate inputs and outputs.

4.2.2 Modelling implementation of the Simulation Model

To complete the SCTIOTS model for simulation combined MBT, the simulation model must be built to model complex data structure and manipulations. The simulation tool selected is a microscopic railway simulator which can build models of essential elements of various types of TCS, such as ETCS and CBTC. The simulator is written in Java and has been developed at

the University of Birmingham over a period of more than seven years, where it has been utilised in the virtual railway laboratory at the Birmingham Centre for Railway Research and Education (BCRRE) [122]. The feasibility and correctness of modelling using the simulator have been proven by the project Developing and Evaluating Dynamic Optimisation for Train Control Systems (DEDOTS) [123]. Using the library provided by the microscopic railway simulator, the I/O behaviour of a wide range of elements in TCSs, including the OBU, RBC and infrastructure components, can be simulated. During an MBT run, the simulation model performs calculations based on the data collected from the simulated environment and the abstract input received from the abstract model. Afterwards, it sends the required calculation results to the abstract model via internal I/O channels. An interface has been built to enable the translation between the simulation I/O and the abstract I/O, and this is introduced in section 4.4. According to the needs of different tests, different simulation models can be built in the simulator at varying levels of detail. In contrast with the abstract model, where system behaviour is modelled on an abstract level, system behaviour is modelled on a concrete level in the simulation model, which means that more detailed information can be included. This is a significant step for testing complex SUTs like TCSs, because the increased detail in the model can lead to a more accurate testing result.

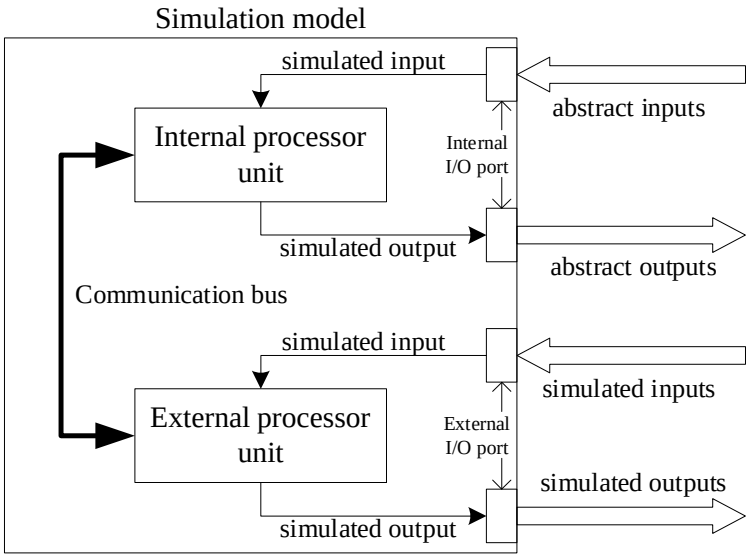


Fig 22 Internal structure of implementation of the simulation model

According to Fig 19 in section 3.2.3, the internal structure of the simulation model is designed, as shown in Fig 22:

As indicated by Fig 22, the simulation model mainly consists of the internal processor unit and the external processor unit. The internal processor unit is responsible for processing the inputs derived from the abstract model and returning the required outputs. The external processor unit is responsible for processing the simulated inputs from the external environment and returning the required simulated outputs to the model. Translation of the data format is realised by the internal and external I/O ports, and processes inside the simulation model are all based on the simulation data type. Communication between the internal and external processor units can happen via the communication bus when necessary. Therefore, the outcome of an abstract input can be a processed result involving an interaction between the internal and external processor units.

4.3 Test Tool

With the specification model obtained, the next important step is to apply the test tool to generate the inputs required for black-box testing and to inspect the outputs derived from the input executions. To achieve these automatically, the test tool needs to extract the I/O sequences from the specification model and to determine the consistency between the SUT and the specification according to the conformance relation. TRON is used to realise online MBT because it is compatible with the model in the TIOTS format, which is built on UPPAAL. TRON provides the online test algorithm which generates, executes and inspects the test simultaneously by connecting with the SUT. During the test implementation process, the outputs generated by the input executions are collected by TRON and compared with the outputs required by the specification model [99]. Because of the characteristics of online MBT, the test tool only needs to consider the next reachable set of symbolic states, $RS \subseteq S \times E$, which is based on the current set of states that are occupied in the specification model after a timed trace is executed. If the observed output complies with the expected output, the test tool will accept the observed output and move on to the next states which belong to the reachable set [110]. Based on the defined SCTIOTS, only the abstract model of the specification model needs to be analysed by TRON. Therefore, the test tool TRON can be adopted as the test generation engine in the implementation of simulation combined MBT without modifying its internal composition and structure. The following pseudo-code describes the operation principle of the online MBT algorithm integrated in TRON [107, 124]:

Algorithm 1.

Initial: $RS \coloneqq \{(s_0,\ e_0),\ clock = 0\}$

while $RS \neq \Phi$ and $clock \leq delay$

do *choose randomly*:

Action:

if $Input(RS) \neq \Phi$

$randomly\ choose\ \ a_I \in Input(RS)$

$send\ \ a_I\ \ to\ SUT$

$RS \coloneqq RS After\ \ a_I$

Delay:

$randomly\ choose\ d \in Delays\ (RS)$

$wait\ for\ d\ or\ activated\ by\ output\ \ a_O$ if $d' \leq d$

if $a_O\ \ arrives$ ***when*** $d' \leq d$

then

$RS \coloneqq RS$ ***After*** $\ a_O$

if $\ a_O \notin Outpu(RS)$ ***then return*** $fail$

else $RS \coloneqq RS$ ***After*** $\ a_O$

else $RS \coloneqq RS$ ***After*** d

Restart:

$RS \coloneqq \{(s_0,\ e_0),\ clock = 0\}$

reset SUT ***if*** $RS = \Phi$

then return $fail$

else return $pass$

The algorithm illustrates the core operational principles of the online MBT test tool, TRON. During the initialisation process of the algorithm, TRON chooses one action from the three optional ones which are: a. randomly choosing a valid input from the current input set and sending it to the SUT; b. opening the output observation channel by choosing a legal amount of delay; c. resetting the SUT and restarting a new cycle. To cover as many possibilities as possible, TRON continuously repeats the process until inconsistency between outputs is detected, drawing a 'Failed' conclusion, or the testing time expires without finding

consistency, drawing a ‘Passed’ conclusion.

From the description of TRON above, traditional application of the test tool is based on the fact that the entire abstract models of the SUT and its operational environment can be fully obtained; this is not achievable for the application scenario in this thesis. According to SCTIOTS modelling theory, part of the system behaviour is held within the simulation model and using the simulator’s data format. The SUT is embedded in the simulation environment to realise HIL testing during the testing procedure. Therefore, the test tool TRON should be capable of interacting with the simulation model and the HIL environment in implementation of simulation combined MBT, which can be realised by the following structural design:

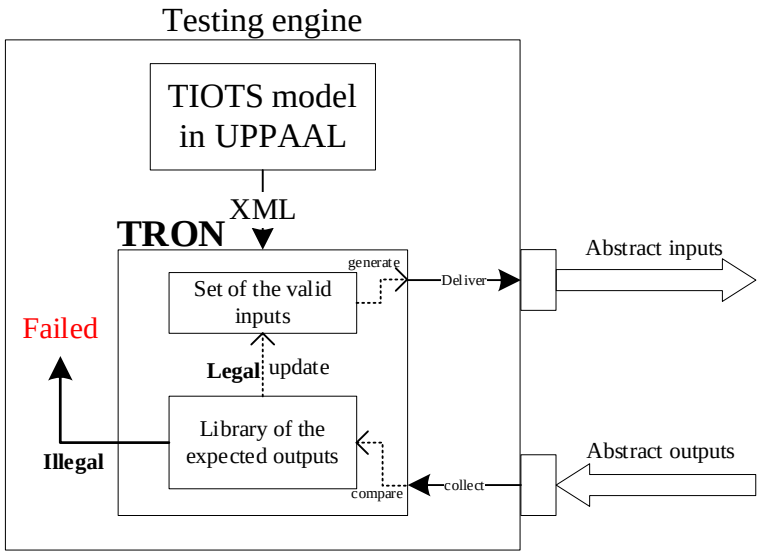


Fig 23 Internal structure of the test tool UPPAAL-TRON

As indicated by Fig 23, the test tool TRON extracts the valid inputs and expected outputs from the loaded TIOTS model in XML format. Based on the logic defined in **Algorithm 1**, the inputs are chosen to be executed, and derived outputs are collected and compared with the

expected ones. Until an illegal output is found by TRON or the testing time expires, the search loop will keep running.

4.4 I/O Sequence Manager

In keeping with the earlier description of the simulation combined MBT method, the abstract model should interact with the simulation model internally and interact with the IUT externally. Therefore, synchronisation needs to be established between internal and external interactions to avoid overwriting data, and causing issues in the I/O sequence. The inputs generated by the test tool and the outputs that can be recognised by it are all in the abstract format determined by the TIOTS model built in UPPAAL. As a result, to connect the test tool and the SUT or the simulation model, the inputs and outputs need to be dynamically translated between the abstract format and the simulation format during the test implementation process. To achieve this, an I/O sequence manager has been designed to realise the synchronisation relations between different I/O channels and to transform the inputs and outputs into the required formats. The manager is written in Java so that it is compatible with the simulation model and the microscopic railway simulator. The overall structure of the I/O sequence manager is presented in Fig 24:

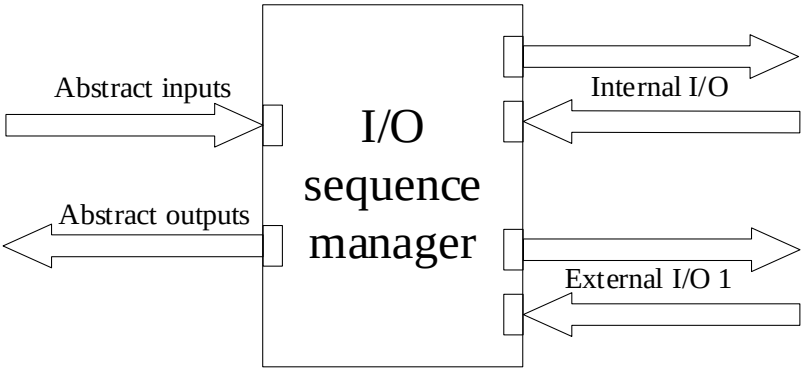


Fig 24 Operation principle of the I/O sequence manager

As indicated by Fig 24, three types of I/O channel are designed in the I/O sequence manager to realise data communication between the test tool, the simulation model and the SUT (within the HIL environment). The abstract I/O channel is designed for delivering abstract inputs generated by the test tool and collecting abstract outputs for test verdicts. The internal I/O channel is designed for transferring internal inputs in the delivered abstract inputs to the simulation model and collecting derived outputs, which are all in the simulation format. The external I/O channel 1 is designed for transferring external inputs to the SUT (within the HIL environment) in the delivered abstract inputs and collecting derived outputs, which are all in simulation format. To guarantee that the test result is correct, every output collected must be derived from execution of the correct input, which means only one input can be delivered at once, and no other inputs should be delivered before the corresponding output is collected. To illustrate the functional logic of the I/O sequence manager, the flow chart in Fig 25 is presented.

As illustrated by Fig 24 and Fig 25, the I/O sequence relations between the abstract I/O actions, internal I/O actions and external I/O actions are managed by the I/O sequence

manager. After an abstract input is generated by the test tool, it is first delivered to the I/O sequence manager to determine the type of input action based on a predefined input library. Based on the input type determined, the corresponding I/O channel is assigned to the input to guarantee that it is dispatched to the correct terminal, which can be the simulation model via the internal I/O channel, or the SUT via external I/O channel 1 and the HIL environment. Meanwhile, the unused I/O channel is blocked to avoid data being overwritten. Once the input is observed to arrive, the input channel between the test tool and the I/O sequence manager is blocked to avoid the next input arriving before the output derived from the last input execution is collected. After the abstract input generated by the test tool is delivered to the simulation model or the SUT, the input channel of the internal I/O channel or external I/O channel 1 is blocked, and the corresponding output channel is activated to wait for output collection. After the output is sent from the simulation model or the SUT, it is collected and delivered to the test tool to identify whether it is valid or not. If the output is correct, the testing process will continue by generating the next input. If the output is incorrect, the test will be terminated with a failed test verdict.

Fig 25 Flow chart of the functional logic realised by the I/O sequence manager

During the testing implementation procedure, the format of I/O actions needs to switch

between the abstract format which can be recognised by the test tool and the simulation format which can be recognised by the SUT or the simulation model; this is made possible by the I/O sequence manager. The I/O sequence manager is not only a controller for handling the sequence of opening and closing the two types of I/O channel, but also an interface for mapping abstract I/O actions and simulation I/O actions. It should be noted that external channel 2 in Fig 20 is not influenced by the I/O sequence manager, so that the simulation model can communicate with the HIL environment and the SUT periodically without interruption. External I/O channel 2 is out of synchronisation with external I/O channel 1 and the internal I/O channel because the I/O exchange period via external I/O channel 2 is significantly faster than the ones via the internal I/O channel and external I/O channel 1, which do not translate between the abstract format and the simulation format. As a result, isolating external I/O channel 2 from the other two types of channel is helpful for improving the operating efficiency of the entire testing process and reducing the design difficulty of the I/O sequence manager.

4.5 HIL Environment

As shown in Fig 24, the IUT is integrated with an interface and does not directly communicate with the test tools or the I/O sequence manager. Since a TCS is a complex integrated system containing wayside, on-board and communication-related equipment, testing individual components such as the EVC or RBC needs a corresponding testing environment because these individual components cannot work independently from their operational environment. Therefore, the HIL environment is designed to integrate the

individual SUTs in a simulated operational environment, making them work as they would in real operational environments. The simulated environment for SUTs is easier to reconfigure when testing different SUTs without the extra costs of using a real testing environment. The microscopic railway simulator introduced was used as the modelling tool for building the HIL environment. Since the simulation includes almost all the essential components in the different types of TCS, it can be easily transformed into an HIL environment by removing the simulated components representing the real SUT. Because it is a simulator written in Java, it has decent compatibility with other Java programmes such as the simulation model and the I/O sequence manager. Fig 26 presents an example of the environment for HIL testing, which is established by the microscopic railway simulator. Depending on different SUTs, the following elements can need to be modelled by the simulator.

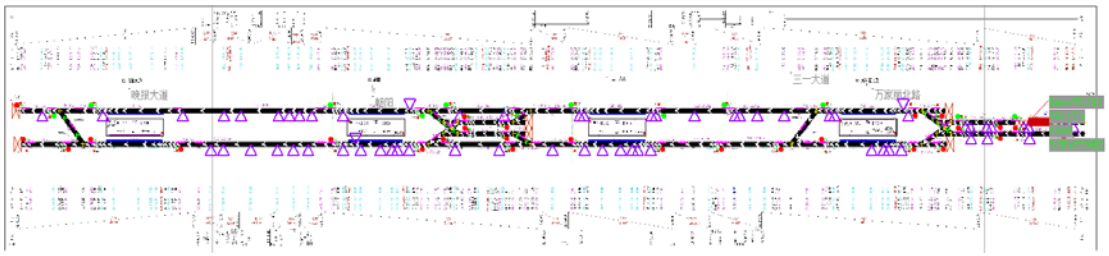


Fig 26 Schematic of the testing scenario for a single train

- Vehicle Model

The vehicle model simulates real trains running on a real track. As the vehicle is the controlled subject of the SUT VOBC, the factors influencing train movements must be included in the simulator. These factors include the train's maximum speed, number of coaches, total vehicle length and weight, and the relations of the traction/resistance force and

the vehicle speed, etc. Some physical factors which require an enormous amount of experimental data to model, such as the friction between the vehicle wheels and the track surface, or the extra resistance caused by extreme weather, are not considered in the simulator. This is appropriate as these kinds of factor are not the focus of this thesis.

- Infrastructure Model

The infrastructure model is a key component of the environment model, and is necessary for TCS operation so that it needs to be simulated to generate necessary inputs during the test. Since the simulated infrastructure contains a wide range of different components, only the ones which are closely related to the testing are discussed in this thesis, which include signals, balises, point switches, and axel counters.

- Timetable Model

The timetable model is designed to indicate the time point at which the simulated train should arrive at a certain position, such as a station. All the trains controlled by the microscopic railway simulator should follow the timetable for operation, arriving at the destination in time. In the single-train scenario, the SUT train is controlled by the testing platform but not the microscopic railway simulator. Therefore, the timetable is not included in the testing environment.

4.6 Data flow in the Simulation Combined MBT Platform

In the previous sections of this chapter, the essential elements for implementation of

simulation combined MBT have been introduced. Based on the essential elements explained, the author summarises data transmissions between the components of the platform.

As illustrated by Fig 20, the testing platform consists of the IUT model with a two-model-combined structure, the test tool TRON which is utilised to control the testing process, the I/O sequence manager which is designed to control the I/O sequence, and the HIL environment which is designed to provide a testing environment for SUTs. To explain in more detail the operating principle of the simulation combined MBT platform, data flow through all integrated components in the testing platform is illustrated in Fig 27:

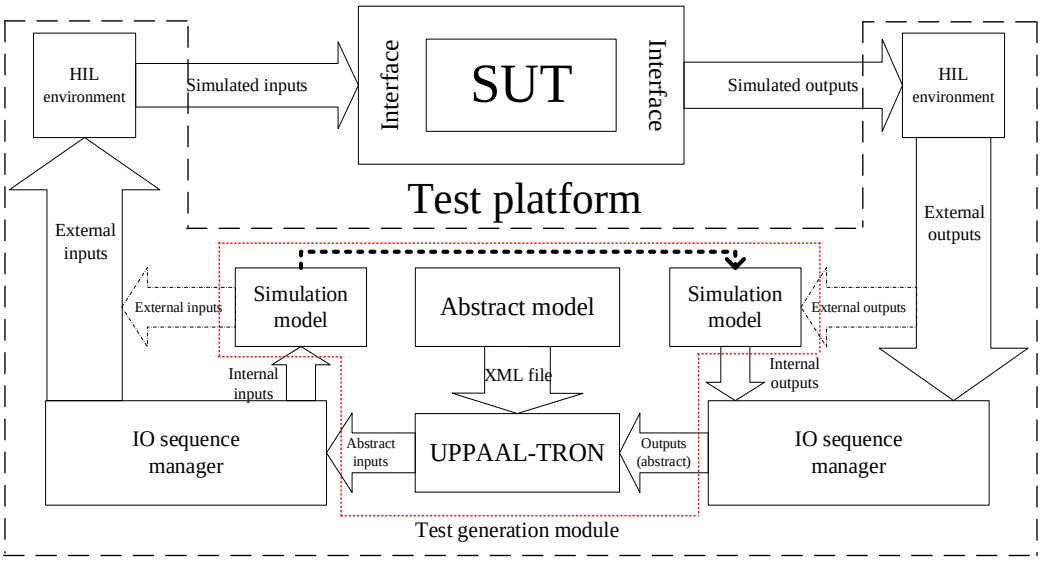


Fig 27 Operating principle of the simulation combined MBT platform

As indicated in Fig 27, the operating principle of the simulation combined MBT platform is explained by illustrating the direction of data flow through components of the testing platform. When the testing process starts, the abstract model is loaded by the test tool TRON in XML

format. By analysing the input model, TRON extracts the currently valid input and records the expected output derived from the input execution. Based on the type of the current input, the I/O sequence manager dispatches the current input to the SUT through the HIL environment via the external I/O channel, or to the simulation model via the internal I/O channel. On the external channel, the input is sent to the SUT and is executed by the SUT. The output generated is translated by the HIL environment and sent back to the test tool TRON via an external I/O channel. On the internal channel, the input is executed by the simulation model, and output is obtained by the test tool TRON. During the whole procedure, the simulation model periodically exchanges data with the SUT via the HIL environment and updates the variable changes for the test tool. The collected output is compared with the expected output desired by the abstract model. If the collected output complies with the specification, the testing process carries on and the next valid input will be tested until the testing time expires. If the output does not comply with the expected one, the testing process will be interrupted, and the testing is finished with a failed verdict. If no inconsistency is found during the whole testing process, the testing is finished with a passed verdict.

5 Functional Testing Case Study on a CBTC System

In this chapter, two cases studies on a typical CBTC system are presents to explain how to apply the proposed simulation combined MBT methodology to undertake functional testing of TCSs. Case study 1 adopted a single train scenario, which is a close to ideal test scenario containing one train, aiming to explain steps to undertake functional testing on the simulation combined MBT platform in details. To study whether the proposed simulation combined MBT is suitable to test complex SUT in realistic scenarios, Case study 2 uses a multiple train scenario with three trains in operation.

As a result of the sharing of functional architectures between CBTC and ETCS, which has been analysed in 2.1, the results of CBTC case studies chosen in this thesis could also be adopted for functional testing in ETCS.

5.1 Case 1: Single Train Scenario

In this chapter, the author applies the presented simulation combined MBT method in the testing of a VOBC which is simulated in the microscopic railway simulator. The real hardware of the SUT is in China but the simulation is built based on its specification. The simulated VOBC is used to realise the functions which are provided by the real hardware and software of the VOBC. Using simulation to replace the real equipment can decrease the risk of damage during the test procedure. An interface is built to transmit data between the SUT and the testing platform, and a communication delay is simulated at the interface to make it similar to real testing. In this case, a single train scenario is chosen as the testing scenario; the

purpose of the case study is to illustrate the detailed process of the testing platform.

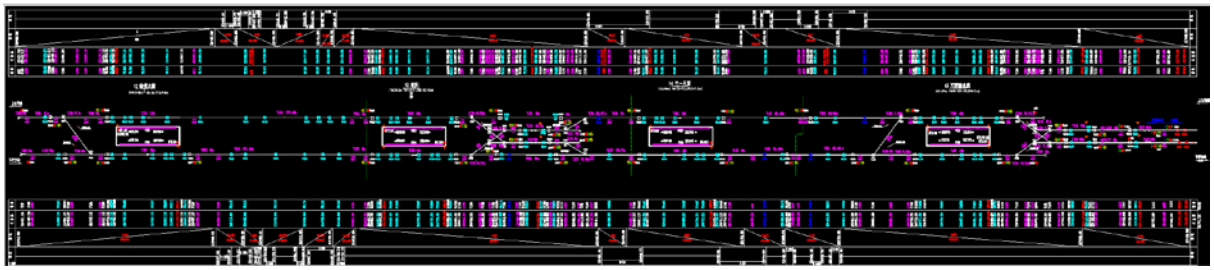


Fig 28 CAD map of Changsha Metro Line 5

The author has selected a test line of Changsha Metro Line 5 which is designed for CBTC system development and testing. The test line contains four stations, three intervals and one simplified depot, as shown in Fig 28. To implement HIL testing, all the necessary elements in the design diagram are modelled in the microscopic railway simulator so that simulated trains can operate in the simulated network as real trains do in the real one. In a single train scenario, only one train runs in the network under the control of the tester.

As indicated in Chapters 3 and 4, the author applies simulation combined MBT by modelling the SUT in two models, the abstract model built by UPPAAL and the simulation model built by the microscopic railway simulator. In this case, the detailed abstract model and simulation model which represent the abstract model and the simulation model, respectively, are presented, and their operating principle is explained. Testing results are recorded, and a testing verdict is drawn based on them.

5.1.1 Abstract Model

As one of the essential parts of simulation combined MBT, the abstract model describes the

key events which happen in the testing process. Associated with the traditional testing process described in Chapter 4, the abstract model plays a similar role to the test case in traditional testing. In traditional testing, the test case is written to specify the test environment, the SUT and the SUT/environment behaviour which should happen during the test procedure. In abstract modelling, the test case is divided into three parts, the SUT, the tester and the communication channels. The SUT and the tester model describe the system behaviour in terms of their interactions. The communication channels are used to describe the potential delays in interactions between the SUT and tester.

5.1.1.1 Specification of the SUT

In this case study, the SUT is a simulated VOBC with the specification of a real one used in the CBTC system of Changsha Metro Line 5. There are a lot of different functions provided by the VOBC, and the author concentrates on overspeed protection in this case. Overspeed protection is a vital CBTC function which protects the train from exceeding the safe speed limit. According to IEEE Standard 1474.1 for CBTC Performance & Functional Requirements, and the simulation specification from the developer of the Changsha Metro Line 5, the VOBC should trigger the EB within 1 second after it receives an overspeed signal (with an allowance of 5 km/h). This specification is associated with the functions of the VOBC, the ZC, signalling and the train, which generate a series of different testing scenarios including different reasons for train overspeed. No matter what factor makes the train overspeed, the VOBC should always comply with the rule that the train's current speed should never be higher than the train's current speed limit. Based on the simulation combined MBT

theory introduced in Chapter 3 and the detailed specification provided by the system developer, the author has refined the SUT specification into the following sub-specifications:

a. The VOBC should receive the train current speed with a period of 200 ms.

b. The VOBC should receive the train MA with a period of 200 ms.

c. The VOBC should calculate the correct speed limit based on the received train MA.

d. In every period, the VOBC should compare the received train speed with the calculated speed limit. If the train speed is over the speed limit, the VOBC should trigger the EB within 1 second.

Since the refined specifications b and c are related to MA, which is calculated using a relatively complex process, they potentially risk increasing the complexity of the abstract model. Therefore, the author has moved these two sub-specifications to the simulation model, and has refined the specification for the TA model as follows:

a. The VOBC should receive the train current speed with a period of 200 ms.

b. In every period, the VOBC should compare the received train speed with the calculated speed limit (from the simulation model). If the train current speed is over the current speed limit, the VOBC should trigger the EB within 1 second.

Once refinement of the testing specification is finished, the abstract model can be built to formalise the testing specification.

5.1.1.2 Abstract Model of the SUT

Based on the refined specification, the author has modelled the test implementation by dividing it into three components, which are the SUT, the tester and the I/O channels. The SUT is the subject that needs to be tested, and its behaviour should comply with the specification. The tester is the person who stimulates the specified inputs and collects the corresponding outputs. By comparing the collected outputs against the outputs expected from the specification, the tester can judge whether the SUT behaviour is correct. The I/O channel is used for the transmission of input and output data.

By combining the refined specification and the expert modelling knowledge of the author , the author builds the abstract model of the SUT, as shown in Fig 29:

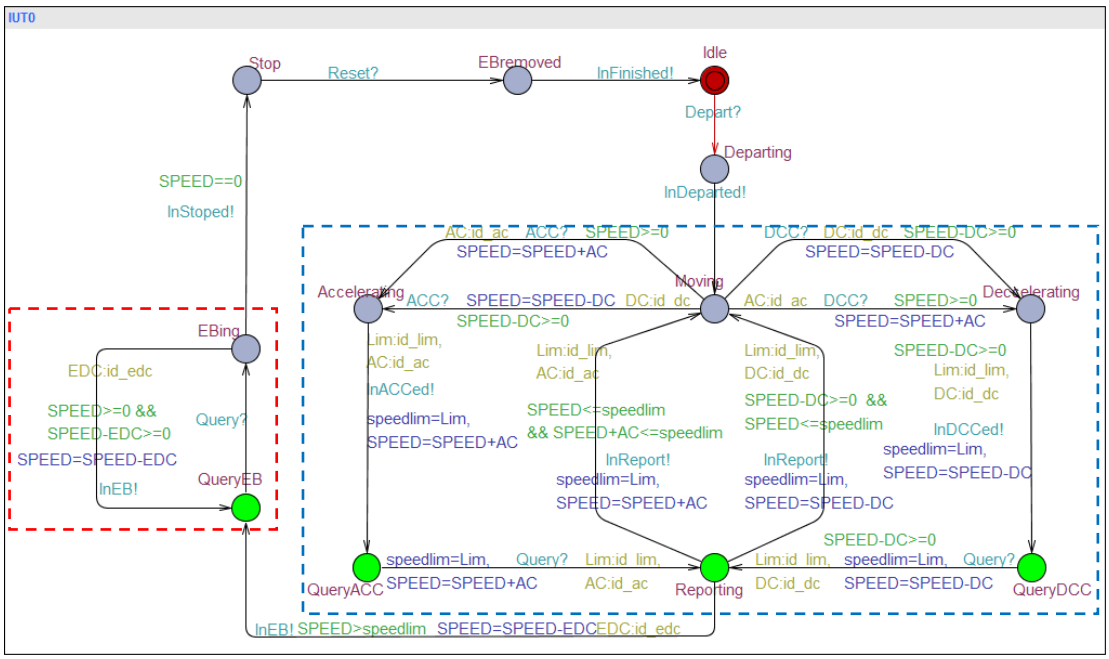


Fig 29 TA model of the SUT for single-train scenario

As seen from Fig 29, the abstract model contains not only the VOBC functions but also some functions provided by the vehicle. In fact, only the states in green and the transitions to these states are related to the VOBC; the other states and transitions are designed to make the train move on the network. In theory, the SUT model should be divided into two models, which are the controller, the VOBC, and the controlled object, the vehicle. However, black-box testing is not concerned about the internal interactions between sub-components. In reality, the tester implements black-box testing and draws a conclusion by observing the train behaviour but not the control command sent by the VOBC. Furthermore, splitting the controller and the controlled object significantly increases the size of the state space and makes the abstract model over-detailed. As a result, the controller and controlled subject are merged, and only external actions are considered. The abstract model of the SUT consists of 12 states and 17 edges, where the states are represented by dots and the edges are presented by arrows.

There are two main parts to the SUT abstract model; the area in the blue dotted box represents the normal moving actions of the SUT, while the red dotted box represents the SUT actions related to the EB. To test the overspeed protection function of the SUT, which is the combination of the VOBC and the vehicle, the first step is to make the train move along the track. This event is realised by sending the command ‘Depart’. After receiving the ‘Depart’ command, the SUT executes it and feeds back a confirmation signal, notifying the tester that the command ‘Depart’ has been received and executed. This event is represented by the signal ‘Departed’ (see section 5.1.1.4). After these two transitions, the SUT abstract model denotes that the train has changed its working condition from dwelling to moving and is ready to

accept further moving commands. In the blue dotted box, the SUT can trigger one of two transitions, which lead to the states ‘Accelerating’ and ‘Decelerating’, respectively. The condition which must be satisfied to make these transitions happen is a value of the variable ‘SPEED’ always no less than zero. The variables ‘AC’ and ‘DC’ are special variables which have random values within a specified range each time the relevant edges are activated. The ranges of ‘AC’ and ‘DC’ are both integers [0,1] which are determined by the communication period and the train’s maximum acceleration and service deceleration. In every period of communication between the tester and the SUT, the variable ‘SPEED’ is observed at least once by the tester. When the maximum train acceleration and service deceleration is $1\ \mathrm{m/s}$ (specified by the system developer), the maximum integer increment or decrement of ‘SPEED’ should be no more than $1\ \mathrm{m/s}$. Therefore, the ranges of ‘AC’ and ‘DC’ can be obtained as [0,1], which means the difference of the continuous two ‘SPEED’ values should not be more than $1\ \mathrm{m/s}$. Each ‘Accelerating’ and ‘Decelerating’ is connected with two different transitions, with consideration of the former conditions. Due to the communication delay between the tester and the SUT, the train can be decelerating or accelerating before the state transits from ‘Moving’ to ‘Accelerating’ or ‘Decelerating’. As a result, the valid value of the current ‘SPEED’ should be in the range of $[SPEED_f - 1,\ SPEED_f + 1]$, where $SPEED_f$ stands for the last value of ‘SPEED’.

After receiving the ‘ACC’ or ‘DCC’ command from the tester, the SUT should feedback a confirmation to notify that the command has been received and executed, which is represented by the signal ‘ACCed’ or ‘DCCed’ (see section 5.1.1.4). Similarly, the variable

values are updated while the transitions are happening. One more variable, ‘speedlim’, shows up, and its value is updated by the special variable. Different from the variables ‘AC’ and ‘DC’, the author lets ‘speedlim’ be a constant value of 22 m/s, determined by the line speed limit from the specification. In a single train scenario, no train is ahead of the SUT, and the MA always extends to the destination of the track. As a result, the train speed limit should follow the line speed limit, which is always 22 m/s along the track of the test line. The benefit is that the possibility space can be significantly reduced without influencing operation of the SUT.

After the SUT sends out the feedback signal, it can receive the command ‘Query’ from the tester, which makes the SUT go to the state ‘Reporting’ and updates the value of the variables ‘SPEED’ and ‘speedlim’ again. In the state of ‘Reporting’, three edges can be activated depending on ‘SPEED’ and ‘speedlim’. If ‘SPEED’ is no more than ‘speedlim’, the SUT goes back to the state ‘Moving’ via two available edges, sending the feedback signal ‘Report’ (see section 5.1.1.4) and updating ‘SPEED’ and ‘speedlim’ one more time. If ‘SPEED’ is greater than ‘speedlim’, it indicates that the train is overspeeding; the EB should be triggered to protect the train so that the SUT sends the signal ‘EB’ (see section 5.1.1.4) to the tester and goes into the red dotted box, which means the train is in the EB condition. Therefore, in the blue dotted box, the SUT continuously accelerates or decelerates until the train overspeeds. In the red dotted box, the SUT continuously decelerates until the train completely stops, which means ‘SPEED’ is equal to zero. Then, the SUT notifies the tester that the train is completely stopped by sending ‘Stopped’ (see section 5.1.1.4), and is ready to be reset by the tester

command ‘Reset’. After receiving the reset command, the SUT executes it and feeds back the signal ‘Finished’ (see section 5.1.1.4), which indicates that a test circulation is finished, and a new one is ready to be implemented.

5.1.1.3 Abstract Model of the Tester

In traditional black-box testing, a tester should inject the specified inputs into the SUT and observe the corresponding outputs. By comparing the observed outputs with the outputs expected from the specification, the tester can judge whether the SUT behaviour complies with the specification. Therefore, the author has built the abstract model of the tester by specifying the commands which can be sent to the SUT, and the responses which can be observed from the SUT with a set of time restraints. One transition can only have one command or response activated, so that the computer can extract an input/output sequence from the abstract model.

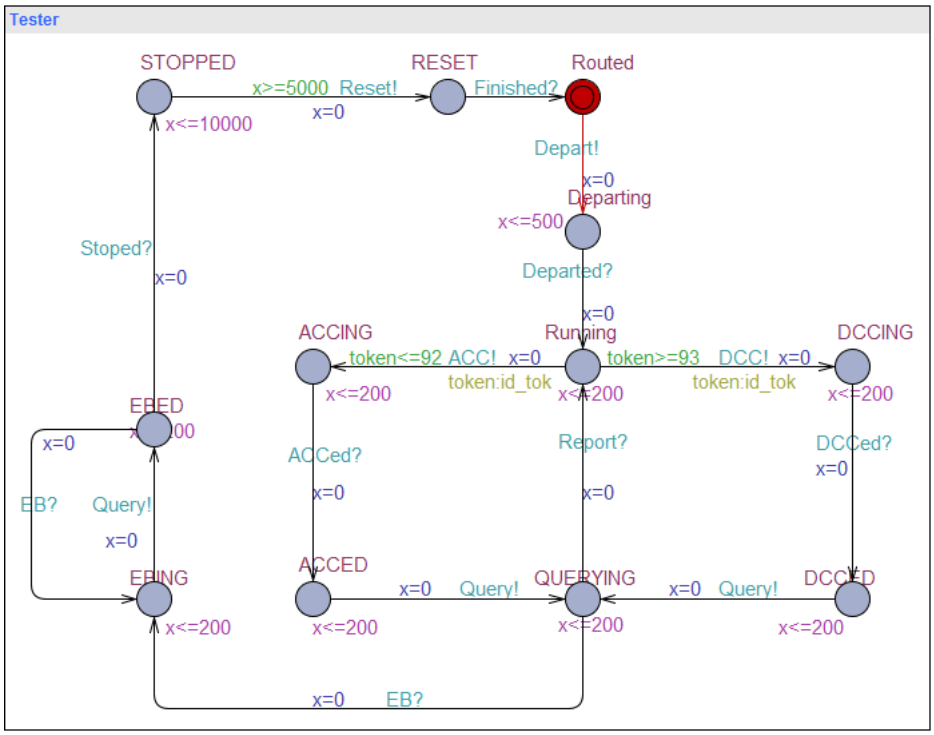


Fig 30 TA model of the tester for single-train scenario

As seen in Fig 30, there are 12 states and 15 edges in the abstract model of the tester, where the model always starts with an input action ‘Depart!’ in its initial state ‘Routed’. Every input action is followed by one of the available output actions which are determined by the current condition of the abstract model. The operation principle has been explained together with the abstract model of the SUT. In the state ‘Running’, two edges can be activated, and the probability of their activation is determined by the special variable ‘token’. The range of ‘token’ is set to be $(0, 99)$, giving the tester a 93% possibility of accelerating the train and a 7% possibility of decelerating the train. The purpose of designing the possibilities of acceleration and deceleration is to make the train tend to travel a longer distance before it exceeds the speed limit, and to make sure that the train can exceed the speed limit before it approaches the end of the track. If the train accelerates without any deceleration, it will trigger the emergency

stop quite soon so that the rest of the track cannot be covered in the testing. On the other hand, if the train decelerates too frequently, it will hardly exceed the speed limit because the train deceleration decreases along with the train speed. Another important parameter in the tester is the time constraints on states and edges, which are used to specify the time-related specification. The time constraint, for example '$x \leq 200$', requires that the system can wait for the next actions for no longer than 200 time-units, and the action must happen when the time limit is reached. Regarding the time constraints on different states, the tester must give a correct input and receive an expect output in time. In theory, there should be another set of time constraints to specify the time relations of the SUT. Since the author has focused on black-box testing, the time constraints of the SUT are again merged into the tester time constraints to reduce the possibility space of the abstract model.

As seen from Fig 30, the abstract model of the tester contains a series of abstract inputs and outputs, where the inputs are represented by '!' and outputs are represented by '?'. The initial action of the tester model is always an input (Depart!), and an input action is always followed by an output action. Table 5 summarises the input and output actions:

Input actions	Output actions
Depart!	Departed?
ACC!	ACCed?
DCC!	DCCed?
Query!	EB?
	Report?
Query!	EB?
	Stop?
Reset!	Finished?

Table 5 Summary of the abstract input and output actions

As shown by Table 5 and discussed in section 4.2.1, all the I/O actions are in an abstract format and cannot be directly processed by the computer. The abstract actions only stand for the I/O channels used for data transmission on transitions. For example, when the SUT model transits from the state ‘Idle’ to ‘Departing’ (which is from ‘Routed’ to ’Departing’ in the tester model), the channel ‘Depart’ is activated, and a command ‘Depart’ is sent from the tester model to the SUT model. This action is assumed to happen instantaneously and to be finished immediately.

5.1.1.4 Abstract Model of the Communication Channels

As mentioned at the beginning of the chapter, a communication delay exists between the tester and the SUT in real testing scenarios. Different conclusions can be drawn from testing if the communication delay is ignored in the abstract model. For example, if the communication delay is non-negligible compared with the time constraints, the testing can draw a fail conclusion because the output arrives too late. Furthermore, nondeterminism can exist due to uncertain communication delays. Therefore, it is necessary to include the communication delay in the abstract model.

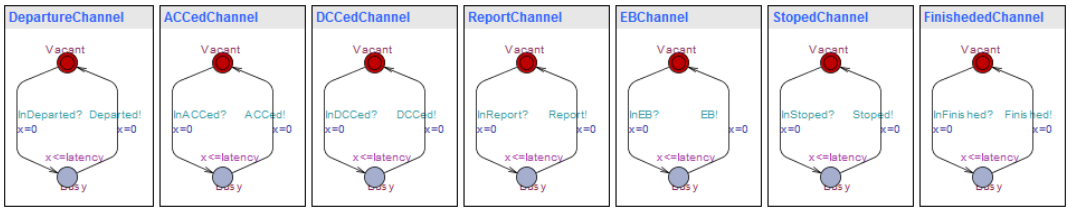


Fig 31 TA model of the communication channels for single-train scenario

As seen from Fig 31, the author has only included the communication delays of the output

channels. In reality, the input delays should be taken into consideration as well. The reason for ignoring the input delays is to reduce the possibility space of the abstract model. Therefore, the input delays are merged into the output delays, which will not influence the test results in black-box testing. According to Fig 31, all the abstract models of the communication delay share the same structure; the only difference is that they respond to different output actions. When an internal output action is given by the SUT model, the corresponding channel is activated and makes a transition from the state 'Vacant' to 'Busy', which equivalently holds the received message for a certain time. When a channel is in the state 'Busy', it cannot be activated again by receiving the message again. After a certain time of the clock 'x' within the time constraints passes, the channel in 'Busy' releases the received internal output by sending out a corresponding external output to the tester. When the clock time reaches the top limit of the time constraints, the channel must release the hold message and send out an external output to the tester. In this case, the time constraint is set to '$x \leq latency,$ where $latency = 20$, to represent that the communication delay should be no more than 20 time-units. This is determined by the communication period of the tester and SUT, which is 200 time-units. According to traditions in TCS functional testing, a communication delay should be no more than 10% of the communication period, to guarantee synchronisation between the tester and the SUT. With the combination of the abstract models introduced, the SUT behaviour in the testing scenario is described to follow the system specification. Fig 32 shows an example of a testing trace contained in the abstract model of the specification:

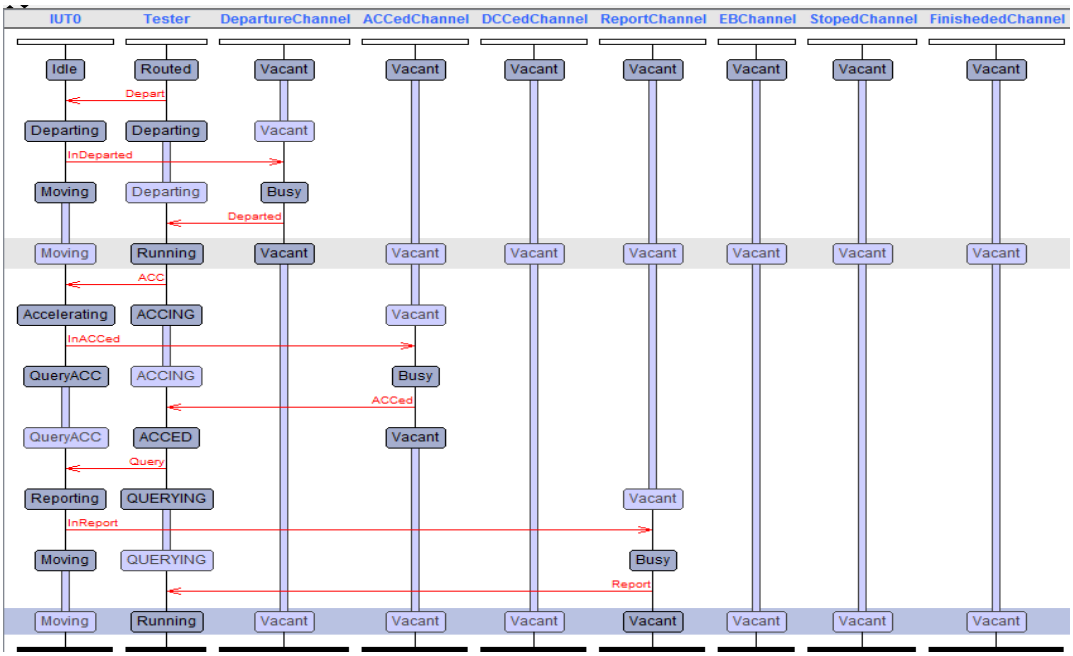


Fig 32 Schematic of a trace generated from the TA model of the specification

5.1.2 Simulation Model

As mentioned in section 5.1.1.1, the author has refined the specification for testing by removing the conditions related to complex calculations. The specification of the VOBC requires that it should be capable of determining whether the train is overspeeding based on the MA given by the ZC. To achieve this goal, the VOBC needs to do calculus to obtain the current speed limit determined by the current MA. The calculation procedure is relatively complex and is not eligible for modelling by the formal method chosen by the author. Simulation provides the solution by simulating the calculation procedure. Therefore, a model of VOBC functional simulation is developed on the platform of the microscopic railway simulator. The following picture is a schematic of typical braking curves for a moving block, which is the foundation of determining speed limit from the MA.

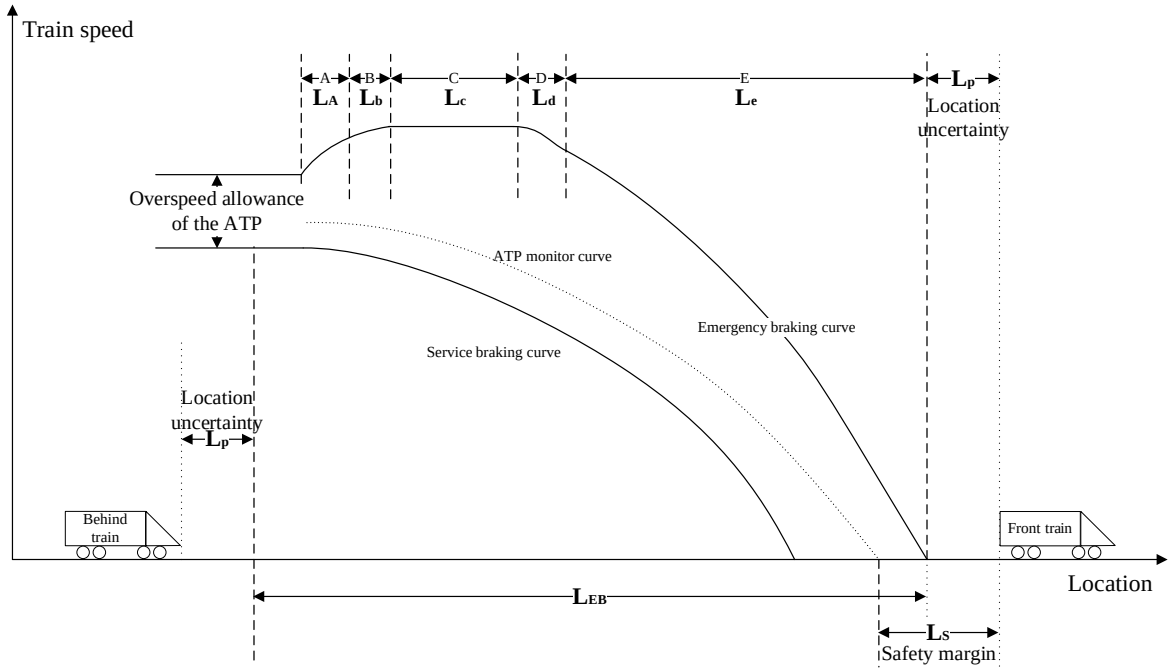


Fig 33 Calculation principle of braking curves

As indicated by Fig 33, the current speed limit of the ‘behind train’ is decided by several parameters: the current distance between the two trains, the physical braking curve of the behind train, and the train position uncertainty of the two trains. The simulation periodically collects the values of these three variables and compares the calculated train speed limit with the line speed limit. The lower value is determined as the current speed limit of the behind train and is transmitted to the abstract model. The following figures explain the VOBC functional simulation procedure:

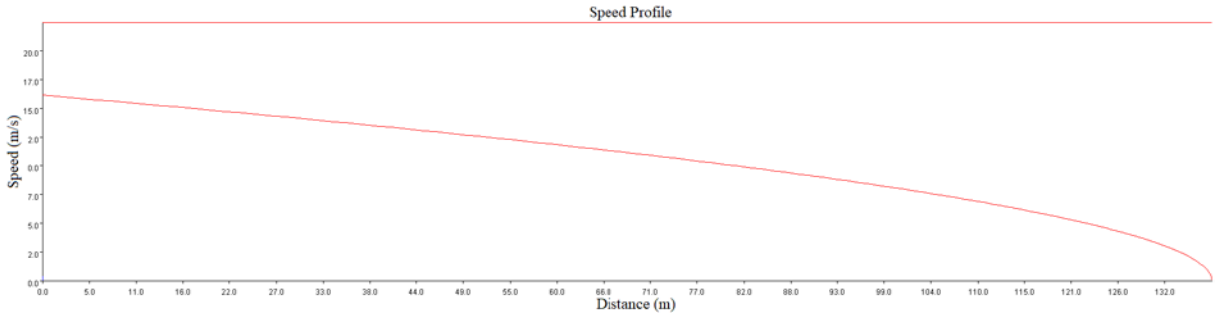


Fig 34 Illustration of the speed limit calculation modelled in the simulator

As shown in Fig 34, the speed limit consists of two main kinds, the static speed limit and the dynamic speed limit. The straight red line at the top represents the static speed limit, the maximum line speed limit which is fixed along with the infrastructure. When the simulation model is activated, it automatically downloads the maximum line speed limit from the infrastructure information provided by the microscopic railway simulator.

The red curve in Fig 35 illustrates the dynamic speed limit which is calculated by the VOBC based on the given MA. A braking curve is calculated with the consideration of several factors including the train parameters, physical laws and the distance between the train and the stopping point. The dynamic speed limit changes along with the changing MA given by the environment. During the testing procedure, the simulation model periodically sends its calculated speed limit to the MBT tool, TRON, and keeps updating the braking curve based on the new MAs.

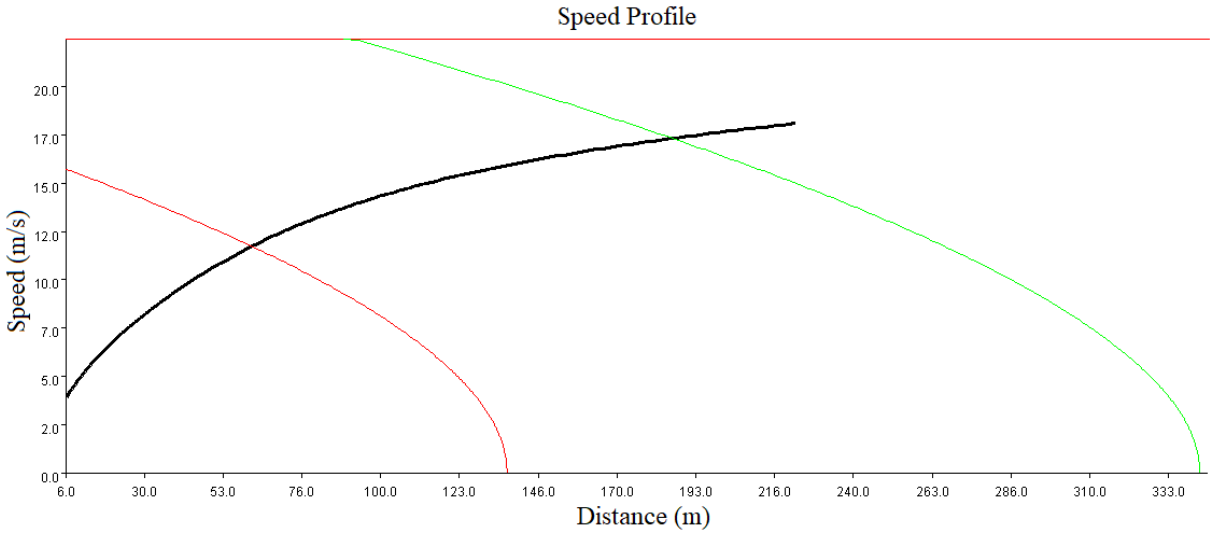


Fig 35 Illustration of the overspeed protection function modelled in the simulator

The two graphs in Fig 36 and Fig 37 depict two different overspeed scenarios in the Intermissive Automatic Train Protection (IATP) mode, where communication between the VOBC and the ZC is interrupted, and the train MA is determined by the signal condition in front. In Fig 36, when the train speed exceeds the dynamic speed limit determined by the MA, the VOBC should trigger the EB and slow down the train according to the braking curve, which is illustrated by the black curve in Fig 36. In Fig 37, when the train speed exceeds the static speed limit, the VOBC should trigger the EB and slow down the train according to the emergency braking curve. Since the SUT VOBC provided only performs EB when overspeed happens, the author has named both kinds of braking as emergency braking.

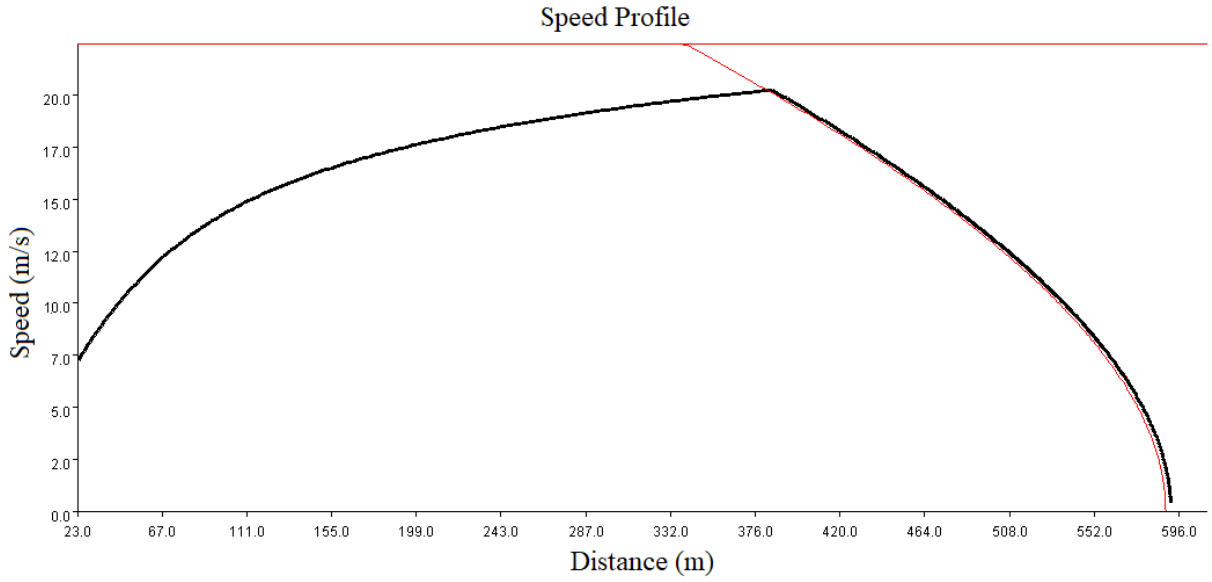


Fig 36 Overspeed scenario: exceeding the speed limit generated by MA

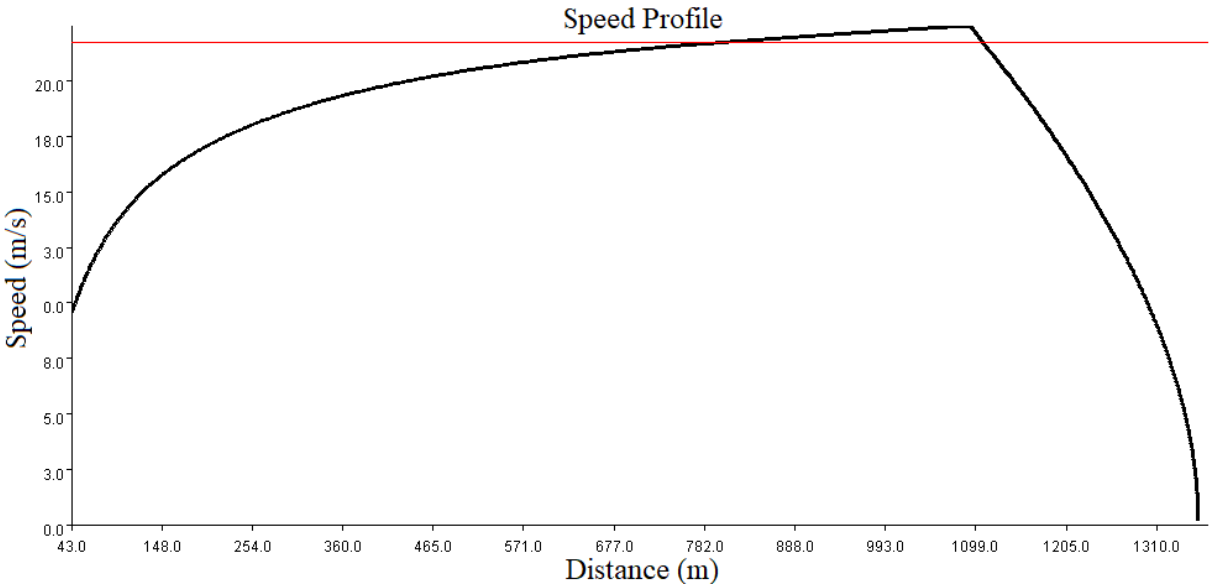


Fig 37 Overspeed scenario: exceeding the speed limit generated by line speed limit

After simulating calculation of the speed limit, the combination of the abstract model and the simulation model can describe the SUT behaviour according to the specification requirements. The two-model-combined structure of the specification model takes advantage of both formal

methods and simulation, simplifying the modelling operation and reducing the complexity of the abstract model. The next step is to build an essential environment for testing, which is modelled by the microscopic railway simulator.

5.1.3 HIL Environment

5.1.3.1 Vehicle Model

According to section 4.5 and the performance parameters provided by the vehicle developer, the simulation of the vehicle adopted in the metro systems is built in the microscopic railway simulator, which is illustrated by Fig 38 and Fig 39 and Table 6.

Based on the figures and table presented above, vehicle movement can be simulated by calculating the train speed and position periodically. During the test procedure, the SUT VOBC controls the simulated vehicle running on the track by detecting the train's movement condition based on the reported train speed and train position. Before going to on-site testing, testing based on the simulated train can decrease the risk of damaging the SUT.

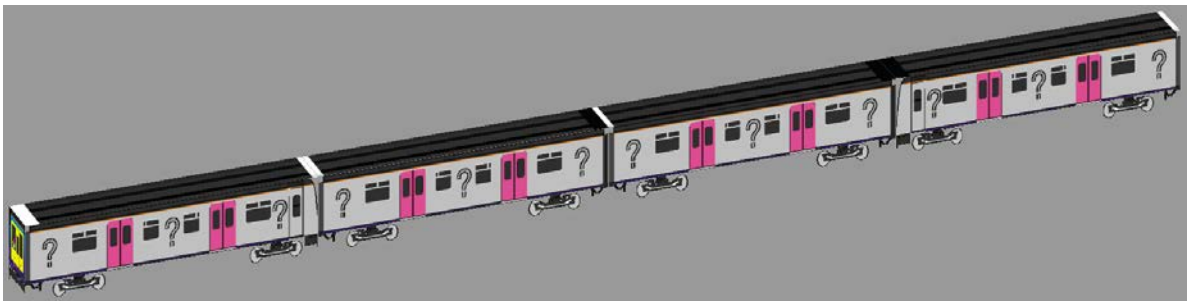


Fig 38 Schematic of the vehicle model in the microscopic railway simulator

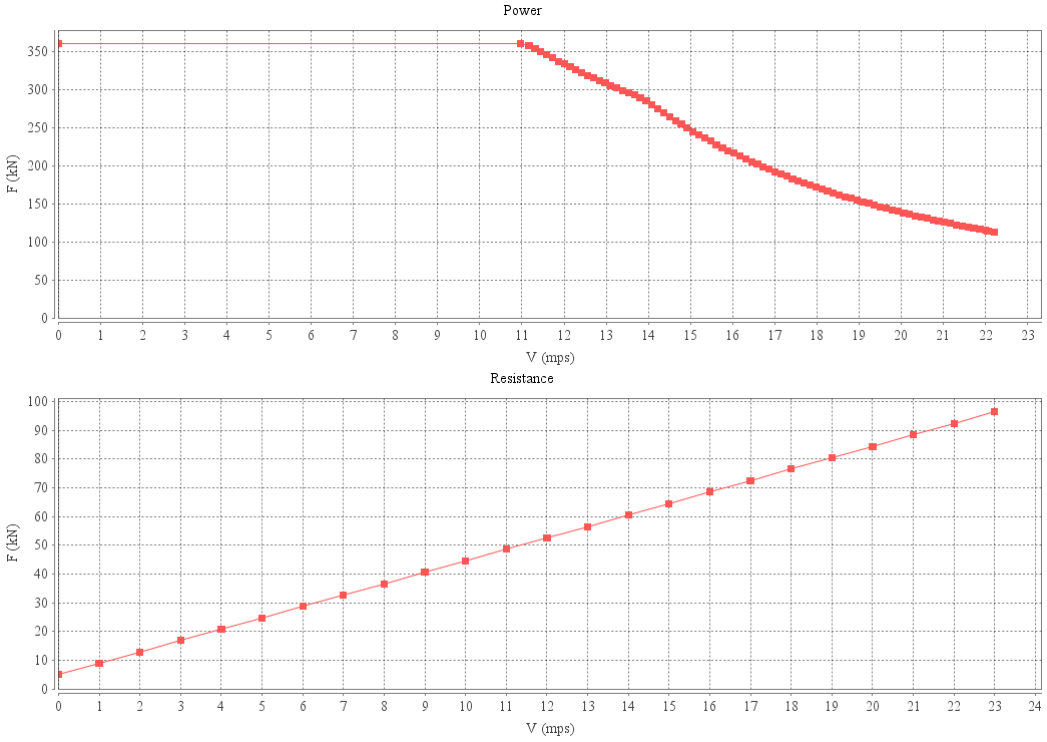


Fig 39 Traction power and resistance power along with various speeds

Description	Maximum speed (km/h)	Coach number	Length (m)	Weight (t)	Type	Maximum acceleration (m/s^2)
Changsha Metro	100	4	114.0	291.6	CBTC	1.1

Table 6 Summary of the parameters in the vehicle model

5.1.3.2 Infrastructure Model

The infrastructure along the track is configured along the simulated network according to the design schematic provided by the system developer, which includes: balises, signals, axle counters and point switches.

- Balises

In this case, the CBTC SUT includes two types of balise, the fixed balise and the variable balise. Fixed balises are designed to inform the VOBC of the train's current location when the train is passing a fixed balise. The VOBC receives a telegraph including the balise ID and balise position sent by the passing balise and determines whether the received information matches with the database. According to the database, the train position will be determined if the balise ID and its position are correct; the train position will be determined as unknown if the received balise ID does not match with its position. As a result, the microscopic railway simulator models the fixed balise by making it send its ID and position when the TIA installed on the train head is approaching the valid receiving range of the balise telegraph.

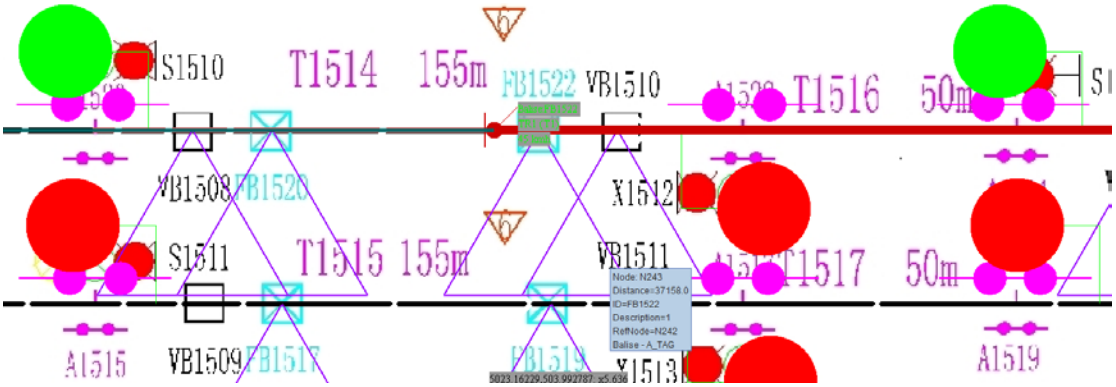


Fig 40 Schematic of a balise-passing event in the simulator

As indicated by Fig 43, the VOBC receives the telegraph of the fixed balise 'FB1522' when the train is passing. According to the system specification requirements, the valid receiving range is set to ± 2.6 m.

Since the variable balise function is irrelevant to the testing in both cases, its function is not simulated in the simulator so that it will not send its telegraph when the train is passing.

- Signals, axle counters and points

In the microscopic railway simulator, the interlocking system is simulated by defining the relations between signals, axle counters and points. When a train is passing a green signal, the axle counter detects the train and informs the interlocking of the train's attendance. Afterwards, the interlocking changes the state of the corresponding signals to inform the following trains that the segment has been occupied. It should be noted that trains at CBTC level do not follow the displayed signal aspects because their movement is determined by the VOBC according to the distance of an obstacle in front of the train, and a 'red' signal is not an obstacle type when the system is operating at CBTC level. When communication between on-board and trackside equipment breaks down, and the CBTC mode is operating in IATP mode, the driver needs to control the train movement according to the signal aspect. However, when the direction of the point needs to be switched, it is necessary to check whether there is any train in the point area. As a result, the interlocking table is included to provide information for the simulator to determine when and whether a point direction can be switched. In Fig 41, a schematic of the signals, balises and points in the simulator is presented.

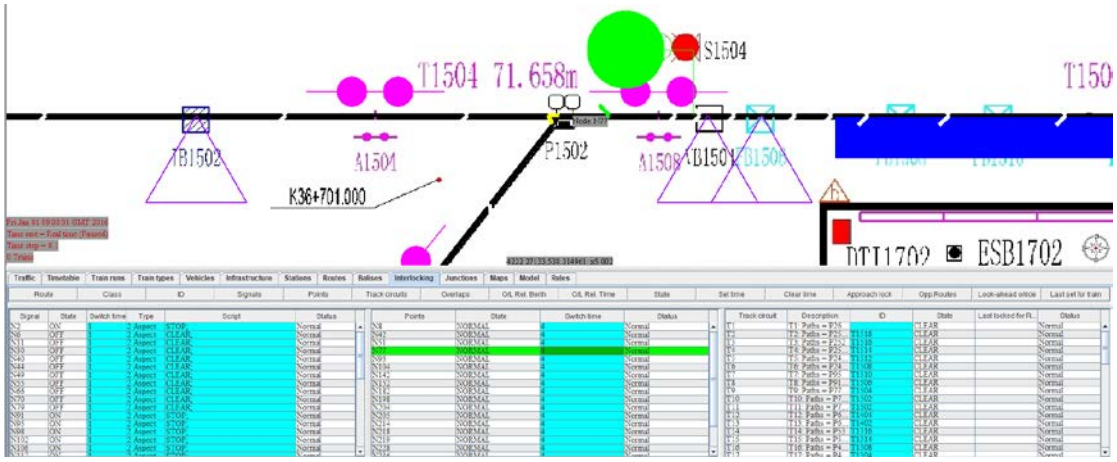


Fig 41 Schematic of signals, axle counters and points of the interlocking in the simulator

5.1.3.3 Timetable Model

In the single train scenario, a timetable is unnecessary because the train does not need to accurately arrive at stations on time. Details of the timetable model can be found in the multiple train scenario, which is in section 5.2.1.2.

5.1.4 I/O Sequence Manager

According to section 4.4, the I/O sequence manager manages the I/O actions which happen on the internal I/O channel and on external I/O channel 1. Therefore, it is necessary to explicitly determine which I/O actions are assigned to which I/O channel, which is summarised in Table 7.

Based on Table 7, the I/O actions involved in the testing are assigned to different I/O channels so that the input generated by the test tool can be correctly sent to the simulation model or HIL environment according to the flow chart in Fig 25. The I/O sequence manager is written in Java, so it can conveniently interact with the simulation model and the HIL environment.

Input	Output	I/O channel	Description
Depart	Departed	External	Train departs when it receives departure command
ACC	ACCed	External	Train executes the acceleration command and informs the tester of the command execution
	EB	Internal	Train executes the acceleration command, which leads to train overspeed, and the VOBC triggers EB
DCC	DCCed	External	Train executes the deceleration command and informs the tester of the command execution
	EB	Internal	Train executes the deceleration command, which leads to train overspeed, and the VOBC triggers EB
Query	PosLost	Internal	The tester queries the train's current operating condition and receives that the train position is lost
	Report	Internal	The tester queries the train current operating condition and is informed of the train's current speed and position
	BaPass	Internal	The tester queries the train's current operating condition and receives that the train is passing a valid balise
Query	EB	Internal	When the train is in the EB condition, the tester queries the train's current operating condition and receives that the EB is being implemented
	Stop	External	When the train is in the EB condition, the tester queries the train current operating condition and receives that the train has been completed stopped
Unlock	Finished	External	After the train is completely stopped by the EB, the tester sends the EB unlocking command and receives that the EB has been unlocked

Table 7 Summary of I/O actions on the internal I/O channel and external I/O channel 1

5.1.5 Testing Results

After configuration of all the elements is finished, the testing is ready to be implemented on the simulation combined MBT platform. In this case, the testing time is set to be 35000 seconds, which means the testing continues until the test expires, or an error is found. Fig 42 presents an example of the testing results, which is part of the train trajectories during the whole testing procedure:

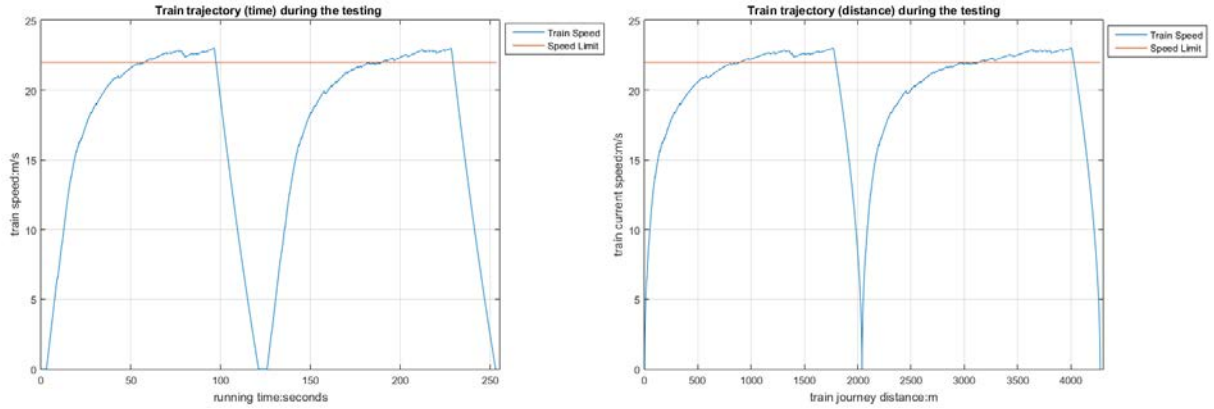


Fig 42 Train trajectory during the testing process: 93% acceleration

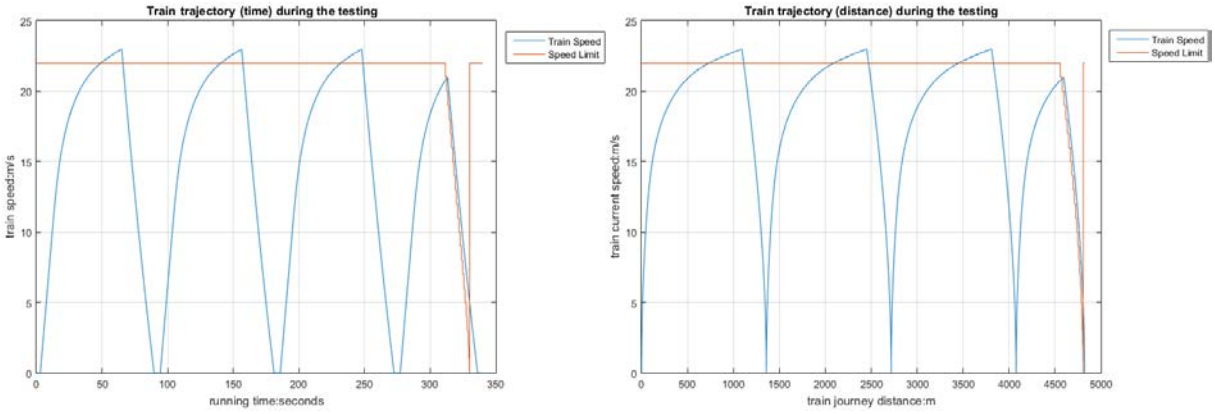


Fig 43 Train trajectory during the testing process: 100% acceleration

As seen from Fig 42, the train triggers the EB twice before it arrives at the destination which is near the end of the track. When the train is completely stopped by the EB and has not arrived at its destination, the testing platform unlocks the implemented EB and the train departs again. When the train is stopped and has arrived at its destination, the testing platform automatically initialises test implementation by putting the train back to its initial position and starting testing again. After the testing period expires, the testing platform interrupts the testing implementation and draws a conclusion on whether the testing is passed or failed. Fig 43 shows the influence of driver tendency. When the driver only accelerates the train, the train

movements are very predictable, and the diversity of the testing result is very poor. The train tends to stop at the same position on the track no matter how many times the testing implementation is run, which can be proven by Fig 44:

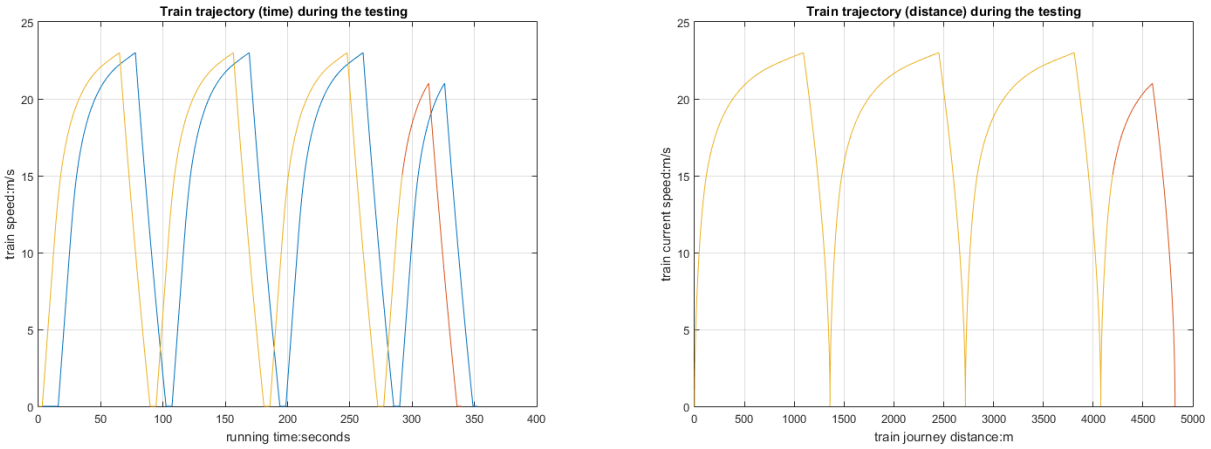


Fig 44 Merged train trajectory run 24 times: 100% acceleration

As can be seen from Fig 44, train trajectories with time do not completely coincide because of the uncertainty of the delay generated by the testing platform or communication delays. However, trajectories with distance perfectly coincide for 24 runs of the testing implementation, which is harmful for covering more possibilities in the testing. As a result, driver tendency is necessary to keep testing coverage.

As revealed by Fig 42, the EB is not triggered immediately after the train exceeds the speed limit. The reason is that the VOBC judges whether the train is overspeeding according to the train speed adding up to an overspeed allowance, which means the VOBC allows the train to keep moving when it is slightly overspeeding. The allowance is designed to avoid the VOBC triggering the EB too frequently in some certain situations. Since a communication delay

exists between the testing platform and the SUT, the platform tends to receive a delayed speed after it finds that the EB is triggered, which means it can receive a speed exceeding the speed limit. Nondeterminism exists for received speed due to the nondeterministic communication delay. Furthermore, the formal methods adopted by the author can only deal with integer data which means that variable differences between the simulation and the testing platform also need to be taken into consideration. In summary, the speed allowance is set to be 1 m/s, which means the EB can be triggered when the train speed is over 82.7 km/h. According to the specification provided by the system developer, the vehicle can guarantee train safety when the allowance is under 5 km/h. As a result, the allowance applied in this case complies with the specification requirements.

The testing implementation is finished after the testing time runs out and a 'PASSED' conclusion is drawn by the testing platform according to the testing results, which is shown below:

```
7301258  TEST: Query()@[34999949602us;34999949602us] at (34999949;34999950) on 1
7301259  TEST: Report(Distance=303)@34999949602us at (34999949;34999950) on 721
7301260  TEST: ACC()@[34999993719us;34999993719us] at (34999993;34999994) on 1
7301261  TEST: ACCed(SPEED=11,speedlim=13)@34999993719us at (34999993;34999994) on 406
7301262  TEST: Query()@[35000005751us;35000005751us] at (35000005;35000006) on 1
7301263  TEST PASSED: Time out for testing
7301264  Time elapsed: 35000005 tu = 35000.005751s
7301265  Time    left: -5 tu = -0.005751s
7301266  Random  seed: 1492204494
```

Meanwhile, the log file recording all the I/O actions is generated by the online test tool TRON, which is shown by Fig 45:

```
  Emulation invariants: IUT_balise_position, Tester_de, DepartureChannel,
                        ACCedChannel,DCCedChannel, ReportChannel, EBChannel,
                        StopedChannel, FinishededChannel.
  Timeunit: 1000us
  Timeout: 35000000mtu
  Inputs:  Depart(), Query(), ACC(), DCC(), Reset(), Unlock(), LosInt()
  Outputs: Departed(), ACCed(SPEED,speedlim), DCCed(SPEED,speedlim), EB(SPEED),
           Stoped(SPEED,Des), Finished(), Report(Distance), BaPass(BaID,Distance),
           BaLost(BaID,Distance), PosLost(BaID,Distance)
TEST: Depart()@[3001653us;3001653us] at (3001;3002) on 1
TEST: delay to (3002 on 2
TEST: Departed()@3002656us at (3002;3003) on 3
TEST: ACC()@[3008672us;3008672us] at (3008;3009) on 1
TEST: ACCed(SPEED=0,speedlim=22)@3008672us at (3008;3009) on 138
TEST: Query()@[3014688us;3014688us] at (3014;3015) on 1
TEST: Report(Distance=0)@3014688us at (3014;3015) on 1024
TEST: ACC()@[3055798us;3056800us] at (3055;3057) on 1
TEST: ACCed(SPEED=0,speedlim=22)@3056800us at (3056;3057) on 138
TEST: Query()@[3062816us;3062816us] at (3062;3063) on 1
TEST: Report(Distance=0)@3063819us at (3063;3064) on 1024
TEST: ACC()@[3104928us;3104928us] at (3104;3105) on 1
TEST: ACCed(SPEED=0,speedlim=22)@3104928us at (3104;3105) on 138
TEST: Query()@[3110944us;3110944us] at (3110;3111) on 1
TEST: Report(Distance=0)@3111946us at (3111;3112) on 1024
TEST: ACC()@[3153056us;3153056us] at (3153;3154) on 1
TEST: ACCed(SPEED=0,speedlim=22)@3153056us at (3153;3154) on 138
TEST: Query()@[3160075us;3160075us] at (3160;3161) on 1
TEST: Report(Distance=0)@3160075us at (3160;3161) on 1024
TEST: ACC()@[3210208us;3210208us] at (3210;3211) on 1
TEST: ACCed(SPEED=0,speedlim=22)@3210208us at (3210;3211) on 138
TEST: Query()@[3217226us;3217226us] at (3217;3218) on 1
TEST: Report(Distance=0)@3217226us at (3217;3218) on 1024
TEST: ACC()@[3264352us;3264352us] at (3264;3265) on 1
TEST: ACCed(SPEED=0,speedlim=22)@3264352us at (3264;3265) on 138
TEST: Query()@[3276384us;3276384us] at (3276;3277) on 1
TEST: Report(Distance=0)@3277387us at (3277;3278) on 1024
```

Fig 45 Fragment of the testing log file

Since the SUT is a VOBC simulation which is designed according to the specification, it is normal that no error is found during the testing process. In Chapter 6, verification of the testing platform is discussed, and its ability to detect error is proven.

5.2 Case 2: Multiple Train Scenario

In Chapter 5.1, the author explained the detailed implementation procedure of MBT online testing based on the simulation combined MBT platform. The testing results preliminarily

prove the effectiveness of the testing platform. However, the testing scenario adopted in that case is a relatively ideal scenario where only a single train operates in the network, which means that the train's MA always extends to the destination of the train and the VOBC in fact protects the train based on the line speed limit. As a result of this, the testing results cannot prove that the VOBC would protect the train when following the dynamic train MA. To completely test the overspeed protection function in the specification, a multiple train scenario was built, introducing more trains into the network to vary train movements. Additional VOBC functions which are relevant to the overspeed protection were added into the specification model to create a more detailed scenario for the SUT. Fig 46 shows the schematic of the multiple-train scenario:

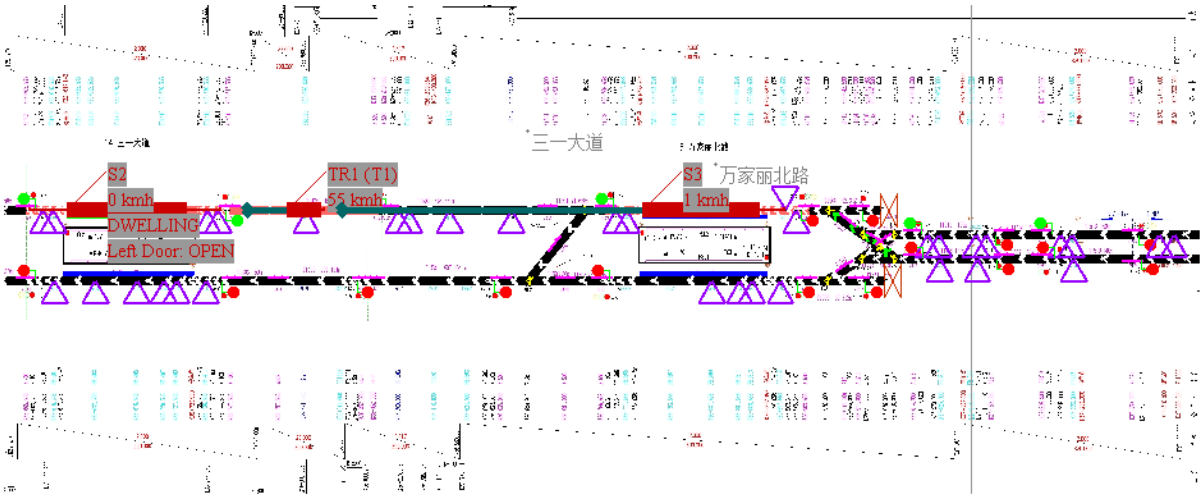


Fig 46 Schematic of multiple-train scenario

As shown in Fig 46, three trains are configured in the same network as adopted in single train scenario. The middle train is the test train which is controlled and monitored by the testing platform. The front train and behind train are both simulated trains, and they are controlled by

the microscopic railway simulator following a designed timetable. These two trains are not installed with the SUT VOBC so that they cannot realise all the functions supported by the middle one. But with the help of the microscopic railway simulator, the two trains can operate safely by following simulated MAs. The main purpose of the configuration is to provide the SUT train with an environment which is more like its real operational environment. In addition to this, the multiple train scenario can test whether the SUT VOBC is safe for whole system operation. In this case study, the author will explain the multiple train scenario by comparing it with the single train scenario, focusing on the differences between the two scenarios and ignoring repeated concepts.

5.2.1 SUT Models and the HIL Environment

To implement testing in the new scenario, the specification model needed to be modified to adapt to the multiple train scenario. At the same time, the HIL environment needed to be re-built to realise the environment for multiple trains. Both elements were evolution of the developed models introduced in the single train case. The train location function of the VOBC was introduced into the testing implementation to improve the operation conditions of the SUT VOBC.

5.2.1.1 Abstract Model

To realise the train location function, the abstract models of the SUT and the tester are modified based on the ones presented in single train case. The updated models are shown by the Fig 47 and Fig 48:

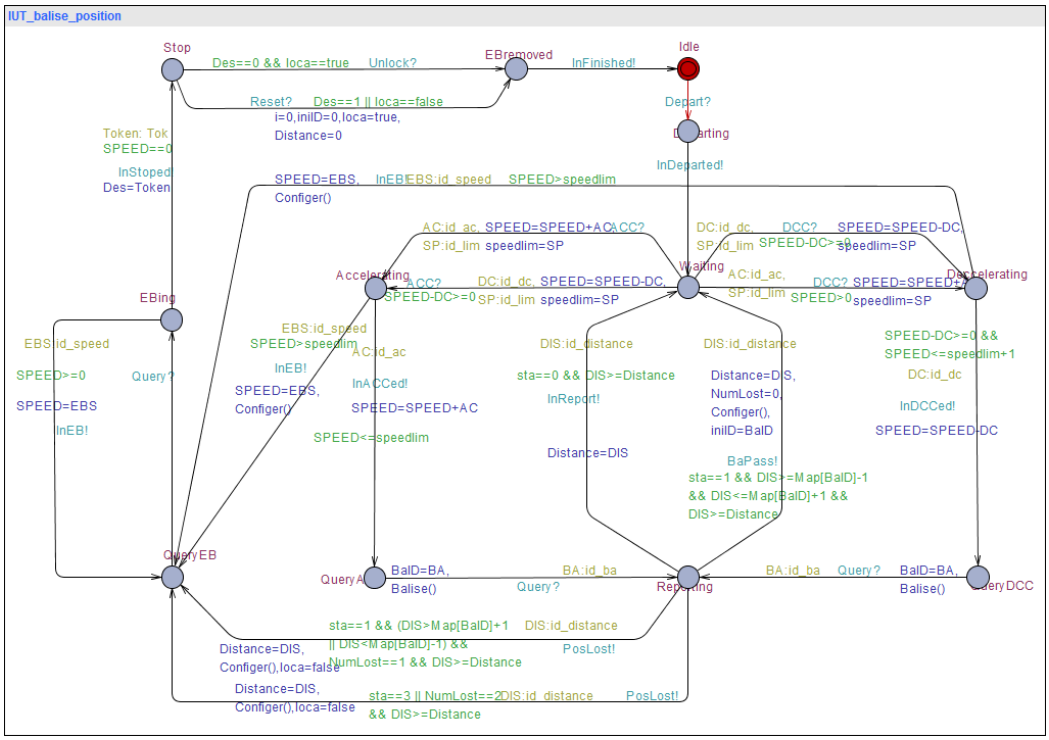


Fig 47 TA model of the SUT for multiple-train scenario

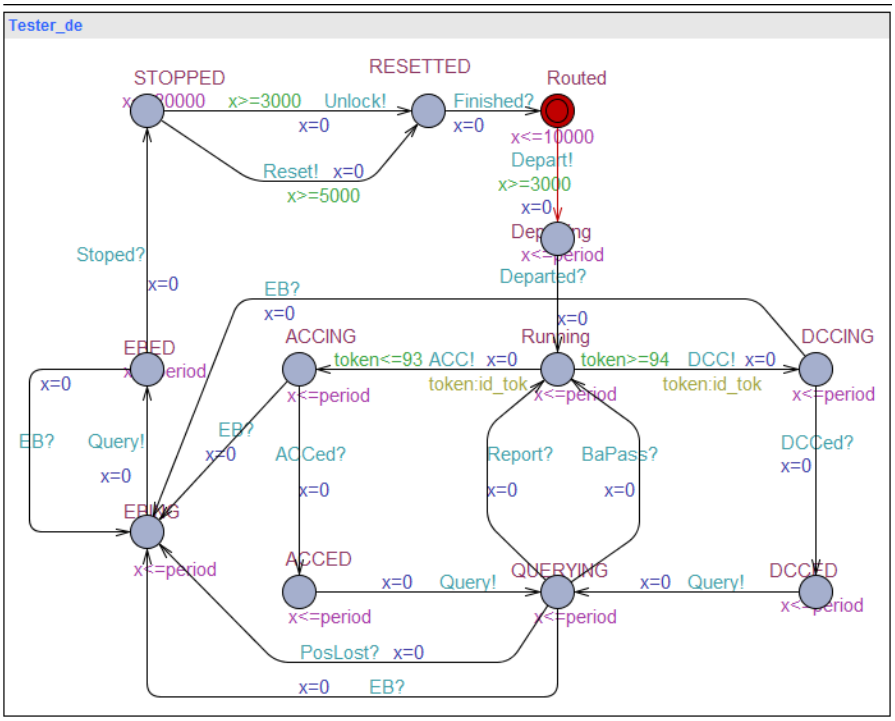


Fig 48 TA model of the tester for multiple-train scenario

As shown by the figures above, the modified TA models keep the general structure of the ones in the single train case. One of the differences is that the train location is realised by adding the variables '**BaID**' and '**Distance**' and the I/O actions '*Unlock'*, '*PosLost*' and '*BaPass*'. Another modification is that the values of the variables '**SPEED**' and '**speedlim**' are checked only on the output action '**ACCed**'. On the output action '**Report**', the variables '**SPEED**' and '**speedlim**' are removed, and the variables '**Distance**' and '**BaID**' are checked by the test tool. The main purpose for the modification is to control the computational load of the test tool, TRON. The testing implementation can fail if the TA model is too complex for the test tool to analyse. If all the four variables ('**SPEED**', '**speedlim**', '**Distance**' and '**BaID**') are checked with the same output action '*Report*', the computational load caused by the combination of the four variable values will be too heavy for TRON to finish the analysis within the time constraints. Therefore, the computational task must be averagely assigned to different I/O actions to avoid the testing implementation becoming unstable.

As presented by Fig 47, the SUT goes to the state '***ACCed***' when '**SPEED**' is lower than '**speedlim**', and it goes to '***QueryEB***' when '**SPEED**' is higher than '**speedlim**'. In the state '***Reporting***', the SUT decides which output action is available, by checking whether the received '**BaID**' matches the received '**Distance**' according to the line map embedded in the VOBC. The output '*Report*' will be sent if no balise ID is received (which means '**BaID**' equals 0). When the SUT receives a balise ID (which means '**BaID**' is non-zero) and the received '**Distance**' is in the valid range (± 5 m) of receiving the corresponding balise ID, the output action '*BaPass*' is activated, representing that the SUT is receiving and accepting a

balise. If '**BaID**' is non-zero but the received '**Distance**' is out of the valid range for the first time, it will be ignored, and the output action '*Report*' will be activated to represent that the balise is not accepted. If two consecutive balises are rejected or the received '**BaID**' is illegal, the output action '*PosLost*' will be activated to represent that the train position is lost by the VOBC, and an EB should be triggered. The code below is embedded in the SUT model and realises part of the VOBC train location logic which determines whether the received balise ID is legal:

```
void Balise(){
if (i<=K-1){
    if (BaID==0) {sta=0;}

    else if (BaID!=0){
        if (BaID==Exp[i]){sta=1;i=i+1;}
        else if (BaID!= Exp[i]){
            if (BaID==Exp[i+1]){
                if (NumLost>=1){sta=3; NumLost=1;i=i+2;}
                else if (NumLost<1){
                  sta=1;NumLost=1;i=i+2;}
             }
            else if (BaID==iniID)
            {
                sta=1;
            }
            else if (BaID!=Exp[i+1] && BaID!=iniID){sta=3; NumLost=0; i=i+1;}
        }
}
}
else if (i>K-1){i=0;}
}
```

Another modification is that '**speedlim**' becomes a variable by setting it to a range of $[0,22]\ \mathrm{m/s}$, which makes the SUT able to accept various speed limits generated by the changing train MA. The variable '**speedlim**' is another reason that the author reconfigured the structure of the SUT model to reduce the possibility space in certain states. The final modification of the SUT model is that the EB can be released if it is triggered by overspeed. If the EB is triggered by a lost train location, the SUT will be reset and the testing

implementation will start from the initial state. The reason is that the VOBC needs the help of the ZC to recover from the train location being lost, while the ZC function is not included in the SUT model in this case. Since the train location function is realised by the abstract model in the specification model, the simulation model remains the same as in the single train scenario.

5.2.1.2 HIL Environment Model

To realise the multiple train scenario, the HIL needs to provide two more trains which can operate in the network. Since only the middle train is controlled and monitored by the testing platform, the other two trains should be controlled by the HIL environment and be able to operate as normal trains; this is supported by the existing functions of the microscopic railway simulator. The movements of the simulated trains are completely controlled by the simulator based on a defined timetable.

As indicated by Table 8 which shows the timetables for the front train and behind train, the timetables rule the train movements by specifying the time point at which the trains should arrive at a certain position. The timetable also specifies the actions that the train should perform at a certain position, including *PASS*, *STOP* and *NONE*. *PASS* means the train should keep its operating condition and pass the certain point at the specified time point. *STOP* signifies that the train plans to stop at a certain position at a specific time point, and *NONE* means no plan is assigned to a certain position. The microscopic railway simulator guarantees that the simulated train follows the timetable when the conditions are satisfied, which means

that no barrier stops the train or that the time slot is long enough for the train to arrive in time. Except for the simulated trains being controlled by the simulator, the operation principles of the simulated trains are the same as those of the SUT train, which means they share the same kinetic equations and have the same influence on the other components in the network.

Service name	Train description	Start date	End date	Drive type
T1	VOBC train	2016-01-01	2016-01-01	UPPAAL driver
S2	Simulation train	2016-01-01	2016-01-01	Simulation driver

Node	Minimum stop time	Required departure time	Type	Stop ID
N290	--	09:00:00	NONE	--
N82	--	--	NONE	--
N76	--	--	NONE	--
N288	**30**	**09:02:20**	**STOP**	**San Yi**
N50	--	--	NONE	--
N285	--	*09:03:20*	*PASS*	*Chao Yang*
N33	--	--	NONE	--
N283	**--**	**09:05:40**	**STOP**	**Wan Bao**
N14	--	--	NONE	--
N1	--	--	NONE	--

S3	Simulation train	2016-01-01	2016-01-01	Simulation driver

Node	Minimum stop time	Required departure time	Type	Stop ID
N290	**30**	**09:00:50**	**STOP**	**--**
N82	--	--	NONE	--
N76	--	--	NONE	--
N288	--	--	NONE	--
N50	--	--	NONE	--
N285	**30**	**09:02:30**	**STOP**	**Chao Yang**
N33	--	--	NONE	--
N283	--	--	NONE	--
N14	--	--	NONE	--
N1	--	--	NONE	--

Table 8 Timetables for simulation trains built in the microscopic railway simulator

The reason that only three trains are included in the scenario is that three trains are enough to form a moving-block scenario. In essence, the testing goal is to determine whether the SUT VOBC can guarantee the train's operational safety under the TCS of a moving block. With a simulated train in front of the SUT train and one behind, the testing implementation can find out whether the SUT train is at risk of crashing into the front train and whether the behind train can crash into the SUT train in some extreme situation. With all the potential risk factors considered by the testing environment, the testing results become more convincing than those of the single-train scenario.

According to the modified specification model, the HIL environment is required to provide two more variables for the specification model, which are the balise ID and the central position of the balise (in the format of journey distance). The added variables are supported by the existing functions of the microscopic railway simulator, which is shown by Fig 49:

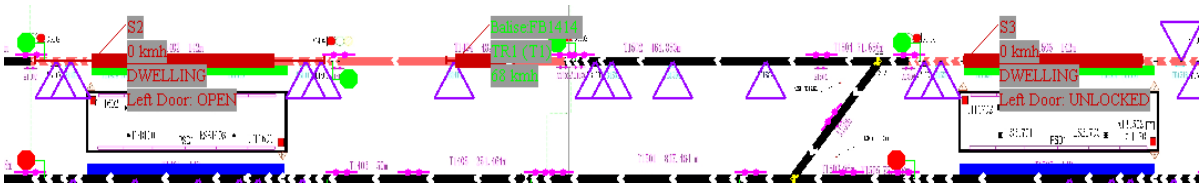


Fig 49 Schematic of the testing environment for train location function

As illustrated by Fig 49, when the SUT train is passing a balise represented by a purple triangle (which means the train head is running into the balise transmission range), the HIL environment sends the corresponding balise ID with the central position of the balise to the SUT train, as displayed by the green label. Based on the received balise ID and central

position, the specification model determines whether the combination is legal and makes the next move. If the received balise ID matches its central position, the SUT VOBC should accept that balise and reset the train uncertainty to zero. If the received balise ID is illegal, or two balises have been missed, the VOBC should trigger the EB because the train position is lost. Without a correct train position, the overspeed protection is meaningless, so that the train location function is a precondition of the overspeed protection function. After modification of the specification model and the HIL environment is finished, the testing platform is ready to execute the testing implementation.

5.2.2 Testing Results

Because the test environment for the multi-train scenario is more complex than that of the single train scenario, the author extended the testing time to 50000 seconds. No failure was found during the testing procedure, and the testing conclusion is shown by the log file below:

```
Options for UPPAAL TRON:
  Search order is breadth first
  Using no space optimisation
  State space representation uses minimal constraint systems
  Observation uncertainties: 0, 0, 0, 0 (microseconds).
  Scheduling latency: 0 microseconds
  Future precomputation: closure(300 mtu).
  Input delay extended by: 0
  OS scheduler: non-real-time.
  Emulation invariants:
               IUT_balise_position, Tester_de, DepartureChannel, ACCedChannel,
               DCCedChannel, ReportChannel, EBChannel, StopedChannel, FinishededChannel.
  Timeunit: 1000us
  Timeout: 50000000mtu
  Inputs:  Depart(), Query(), ACC(), DCC(), Reset(), Unlock(), LosInt()
  Outputs: Departed(), ACCed(SPEED,speedlim), DCCed(SPEED,speedlim), EB(SPEED),
           Stoped(SPEED,Des), Finished(), Report(Distance), BaPass(BaID,Distance),
           BaLost(BaID,Distance), PosLost(BaID,Distance)

TEST PASSED: Time out for testing
Time elapsed: 50000003 tu = 50000.003738s
Time    left: -3 tu = -0.003738s
Random  seed: 1492248172
```

Since multiple trains run in the network, the platform should inspect whether the SUT VOBC can protect the train from overspeed and avoid the train going into a dangerous position where collisions can happen. A crash detection function has been developed on the testing platform to detect whether several train heads or tails appear at the same position on the track during the whole testing process. Fig 50 shows an example of a crash situation happening in the network:

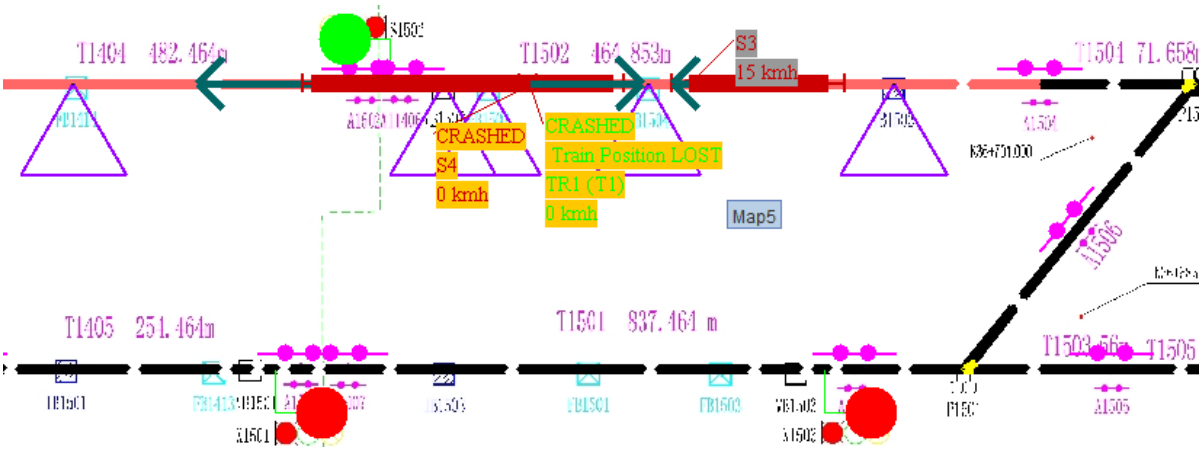


Fig 50 Schematic of the crash detection function

As revealed by Fig 50, the front train crashes into the SUT train because its moving direction is reversed, and its overspeed protection function is artificially removed. Without detecting the SUT train is in front, the front train moves in a normal condition and finally crashes into the SUT train. When the crash happens, the testing platform detects the situation and freezes the two crashed trains to avoid further damage. The operation that causes the train crash is illegal in a real metro system because driving a train in reverse on the track needs a series of preconditions and permissions from other components in the CBTC system. The author uses the assumed case to show the operation principle of the crash detection function. Fig 51

shows part of the testing results, the distance–time graph of the three trains in the network for one run:

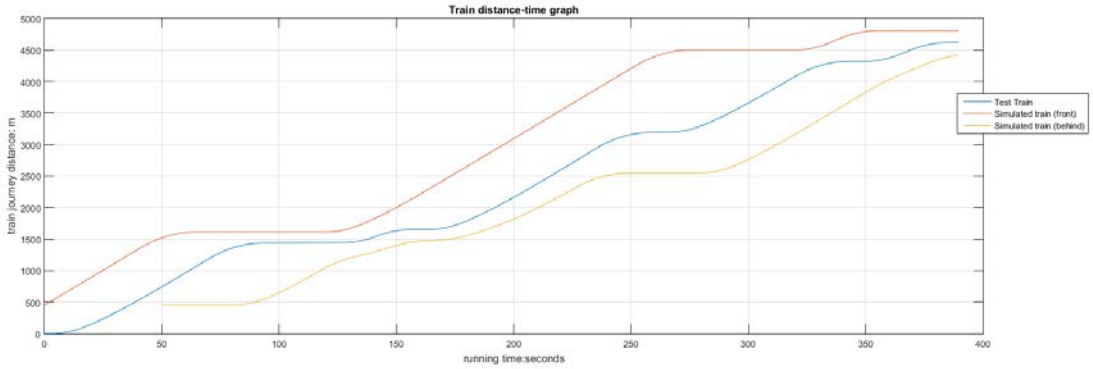


Fig 51 Distance–time graph of the three trains in the network

As can be seen from Fig 50, the distance–time graph clearly indicates that there is no collision happening during the time elapsed. The minimum distance between each two trains can be roughly obtained from the graph; it is about 100 metres, much more than the safety margin which is 40 metres according to the specification. After 50000 seconds running of the testing implementation, no crash was found, and the test was passed successfully.

Another purpose of testing is to find out whether the SUT VOBC can protect the train from overspeed in the multiple-train operation scenario. The following group of graphs record the trajectories of the three trains in one run of the testing implementation:

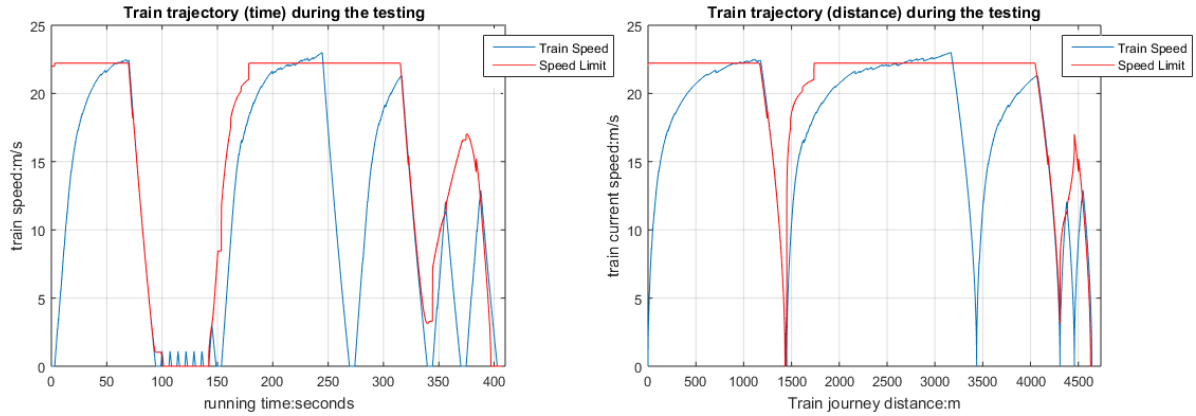


Fig 52 Trajectory graphs of the SUT train in one loop of testing

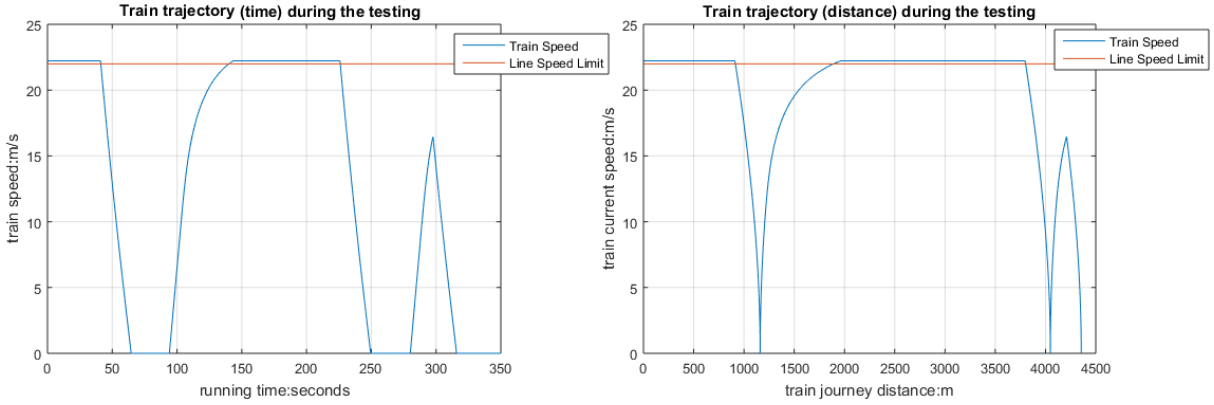


Fig 53 Trajectory graphs of the front train S2 in one loop of testing

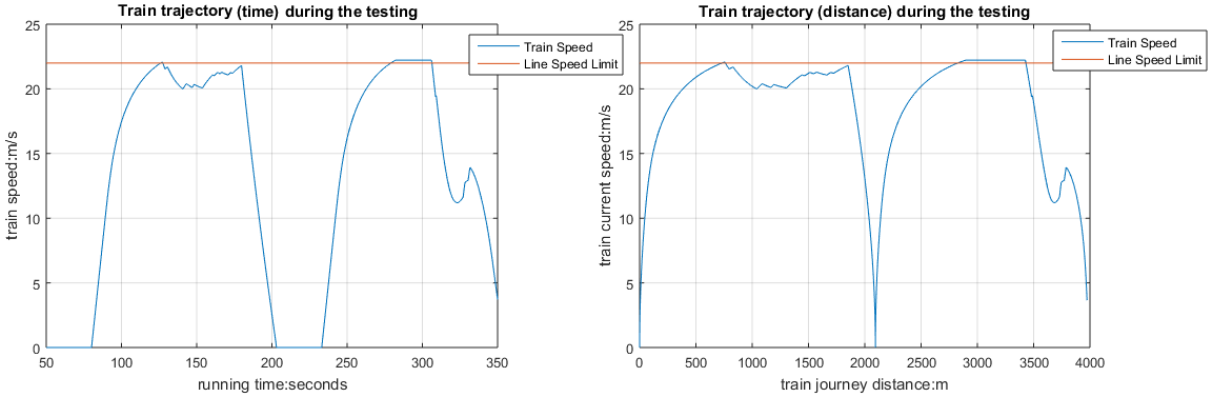


Fig 54 Trajectory graphs of the behind train S3 in one loop of testing

In Fig 52, the SUT train speed tends to follow the various speed limits determined by the

train's MA. The EB is triggered every time the train speed exceeds the speed limit. Since the SUT is the middle train, the speed limits of the front and rear trains are ignored, and only the line speed limit is recorded in their trajectories. When the SUT train arrives at the destination, the testing platform resets the testing implementation by reinitialising the positions of the three trains. Then the testing runs again until the testing time expires.

In the testing scenario, the other reason for triggering the EB is that the train location is lost by the VOBC. Fig 55 shows an example of the EB being triggered by a lost train location:

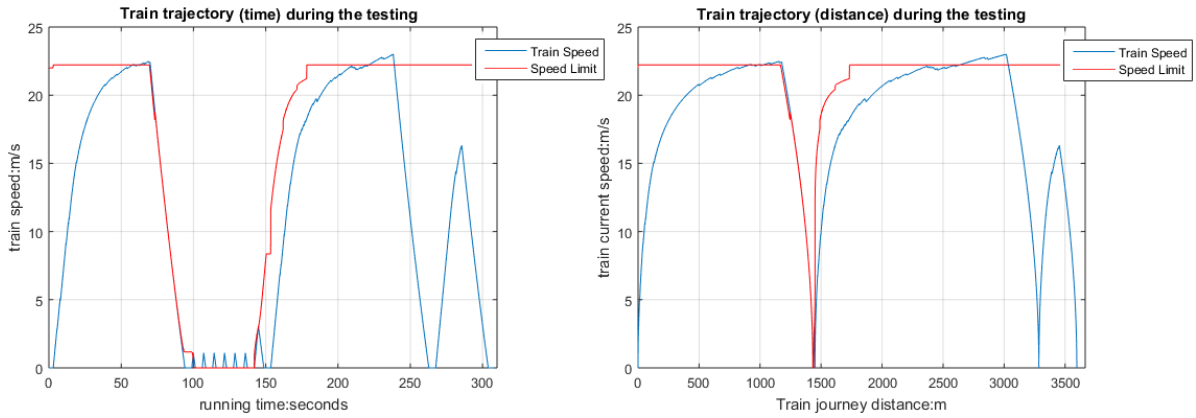


Fig 55 Trajectory graphs of the SUT train for the example of EB due to lost train location

As recorded by Fig 55, the SUT train triggers a third EB after two EB caused by overspeed. The third EB is obviously not triggered by overspeed because the current train speed is far below the speed limit at which the EB is triggered, therefore the EB must have been triggered as a result of losing the train's location. There are various situations in which the SUT VOBC can lose the train's location. For example, if the VOBC has not lost balises before the EB is triggered, the EB is triggered on receiving an illegal balise ID. If there has been an adjacent balise (in front of the current one) lost by the VOBC, rejection of the current balise will

trigger the EB both for an incorrect balise ID, or an incorrect balise central position. In this case, according to the specification model and the HIL environment, the train triggers the EB because it receives an illegal balise ID. Since the HIL environment is not designed to send a wrong balise ID or a wrong balise central position, the reason the EB is triggered is that the second EB triggered by EB forces the VOBC to miss two balises. The following graph describes in detail how the third EB is triggered:

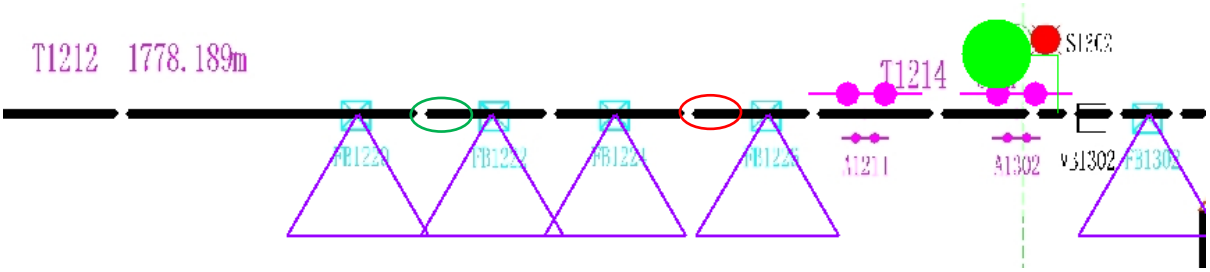


Fig 56 Schematic of scenario in which a third EB is triggered

According to the speed–distance graph in Fig 55, the SUT train triggers the second EB at around 3000 metres, which refers to a point marked in the red circle between the fixed balises ‘FB1226’ and ‘FB1224’. The train eventually stops at the point at around 3250 metres, referring to the point marked with a green circle on the track. Since the VOBC cannot accept balise messages when the EB is being implemented, the two balise messages between the red circle and the green circle are missed. When the train departures again and passes the fixed balise ‘FB1220’, the VOBC determines it has received an illegal balise and triggers the EB for the third time. The deduction can be proven by the log file below and Fig 57:

```
507264  TEST: ACC()@[2260473065us;2260473065us] at (2260473;2260474) on 1
507265  TEST: ACCed(SPEED=22,speedlim=22)@2260473065us at (2260473;2260474) on 626
507266  TEST: Query()@[2260488901us;2260488901us] at (2260488;2260489) on 1
507267  TEST: BaPass(BaID=14,Distance=555)@2260489402us at (2260489;2260490) on 469
507268  TEST: ACC()@[2260527005us;2260527005us] at (2260527;2260528) on 1
507269  TEST: ACCed(SPEED=22,speedlim=22)@2260527005us at (2260527;2260528) on 626
507270  TEST: Query()@[2260543047us;2260543047us] at (2260543;2260544) on 1

508087   TEST: Report(Distance=602)@2271776050us at (2271776;2271777) on 422
508088   TEST: ACC()@[2271814151us;2271814151us] at (2271814;2271815) on 1
508089   TEST: EB(SPEED=23)@2271814151us at (2271814;2271815) on 626
508090   TEST: Query()@[2271831196us;2271831196us] at (2271831;2271832) on 23
508091   TEST: EB(SPEED=23)@2271831196us at (2271831;2271832) on 575

519014  TEST: ACC()@[2323281231us;2323281231us] at (2323281;2323282) on 1
519015  TEST: ACCed(SPEED=16,speedlim=22)@2323281231us at (2323281;2323282) on 506
519016  TEST: Query()@[2323296270us;2323296270us] at (2323296;2323297) on 1
519017  TEST: PosLost(BaID=17,Distance=690)@2323296270us at (2323296;2323297) on 335
519018  TEST: Query()@[2323332367us;2323332367us] at (2323332;2323333) on 1
519019  TEST: EB(SPEED=16)@2323332367us at (2323332;2323333) on 25
519020  TEST: Query()@[2323333369us;2323333369us] at (2323333;2323334) on 1
```

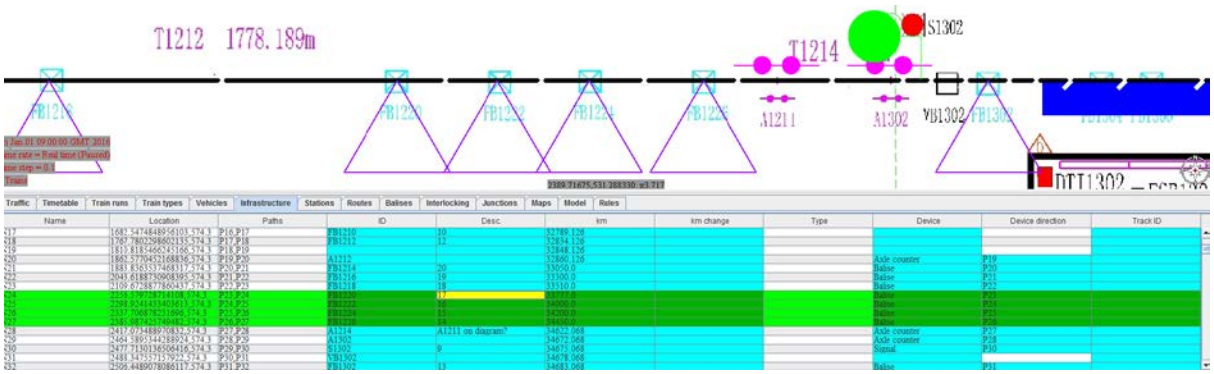


Fig 57 Correspondence relations of balise ID between the abstract model and HIL environment

Based on the log file recorded by the testing platform, the last received balise ID is '**BaID**=14' at '**Distance**=555' (actual distance equals $555 \times 5 = 2775$ m). After receiving balise '14', the train triggers the EB because of overspeed and misses two continuous balises. When the train departs and comes across balise '17', the EB is triggered again because the train's location is lost. Fig 57 indicates the translation relations between the real balise IDs in the HIL environment and the abstract balise IDs (in the 'Desc.' column of the Infrastructure table) in the specification model.

5.2.3 Summary

In this case, the author has extended the prototype of the testing scenario used in the case study in Chapter 0 into an advanced version which is closer to a real testing scenario containing multiple trains travelling on the network. By adding the train location function into the specification model, the prerequisites for overspeed protection are completed. Without a correct train location, the VOBC cannot make a convincing decision on whether the train is overspeeding. The multiple train case provides a relatively complicated environment for the SUT, testing the SUT's ability to protect the SUT train as well as the other trains running in the same network. The testing results of the multiple train scenario are more convincing than those of the single train scenario as the multiple train scenario takes more impact factors into consideration, such as interactions between the three trains. With the help of simulation, more elements can be included in the testing process without decreasing its efficiency due to increased model complexity. By refining complex specifications into abstract format and simulating the rest of the SUT behaviour, the testing platform takes advantage of both MBT technologies and simulation. Although no failure was found during testing, the testing results still indicate the feasibility of the testing platform.

5.3 Conclusion

Two cases were implemented to inspect the testing ability of the developed simulation combined MBT platform. In the single train scenario, the implementation method in Chapter 4 was realised. The author installed the SUT VOBC on a simulated train which can travel on a

simulated network. The overspeed protection function of the VOBC was tested, and the test results indicate that the SUT VOBC complies with the system specification. In the multiple-train scenario, three trains are travelling on the network and only the middle train is protected by the SUT VOBC. The overspeed protection function and the train location function were tested. Since multiple trains travelling on the same line is a necessary operational scenario in CBTC system operation, the author implemented the case to explore whether the developed testing platform is capable of testing the VOBC in such a scenario. The test results indicate that the SUT VOBC can still protect the train from the dangerous situation caused by train overspeed or loss of train location. Since the test scenario was simulated according to real data provided by the system developer, the two cases prove that the testing platform can be applied to test SUTs in an HIL environment. The MBT combined with simulation is proven to be a feasible solution to automate complex SUT testing without the risk of state explosion. Furthermore, the proposed simulation combined MBT decreases modelling difficulties by adopting the simulation model to describe complex system behaviour.

6 Validation and Verification

For black-box testing, only one of three conclusions can be drawn, ***Pass***, ***Fail*** or ***Inconclusive***. However, the three conclusions available cannot be quantised, which means the performance of a test tool cannot be analysed based on the conclusions. The simulation combined MBT platform is a comprehensive testing platform, integrating the formal model in UPPAAL, the simulation model in the microscopic railway simulator, and the online test tool

UPPAAL-TRON. To verify the testing platform, proving the correctness of all the components in the simulation combined MBT platform is necessary, which was done in Chapter 3 and 4. Based on the methodology introduced, Chapter 5 has introduced the simulation combined MBT performed and the testing results obtained. To further prove the effectiveness of the proposed methodology, validation and verification should be implemented.

In engineering field, validation is the process of determining whether the system specification requirements are correctly built to satisfy customer's demands. Verification is the process of determining whether a system is correctly built according to its system specification requirements [125]. Expanding the definition to the field of MBT, specification models should be validated to prove that they have been correctly built to achieve testing purposes and test results should be verified to prove that testing has been correctly implemented to draw valid testing verdict. Therefore, specification models built in Chapter 5 are validated in this chapter. The performance of the testing platform is then analysed in forms of quantised indices according to the specification models validated and the testing results obtained, to verify the effectiveness and performance of the simulation combined MBT platform. Two case studies have been undertaken in Chapter 5 and the author uses the data obtained from the multiple train case study to implement the validation and verification. The multiple train case is used based on the comprehensively rich data when compared with the single train case.

6.1 Validation of the Specification Requirement

In MBT, the specification model represents the informal specification, and guides the computer in the execution of testing implementation. Therefore, the correctness of the specification model is one of the essential components in the MBT. An MBT can draw a wrong conclusion if the original specification is incorrectly modelled. Proving strict consistency between the specification model and the original specification is time-consuming; instead, the solution is to prove consistency within given constraints. As a result, the problem is transformed into the specification model having to comply with the original specification solely within context of the multiple-train testing scenario. According to the discussion in Chapter 5, the specification model consists of the abstract model and the simulation model. Since the abstract model is written in the formal format, it is more likely to contain mistakes made by human factors. The author focuses on validation of the abstract TA model in this chapter. Validation of the simulation model is discussed only briefly because it has been validated in the frame of microscopic railway simulator.

6.1.1 Abstract Model Validation

The abstract model is written in TA format on the modelling tool UPPAAL, which supports model verification by model-checking, a verification technology to automatically and exhaustively inspect whether the model satisfies given properties [93, 126]. To verify the TA model in UPPAAL, the key properties need to be written in first-order logic using the language format desired by UPPAAL. There are four main property formulae supported in

UPPAAL, used to check whether the property P is satisfied by the TA model, which are shown below:

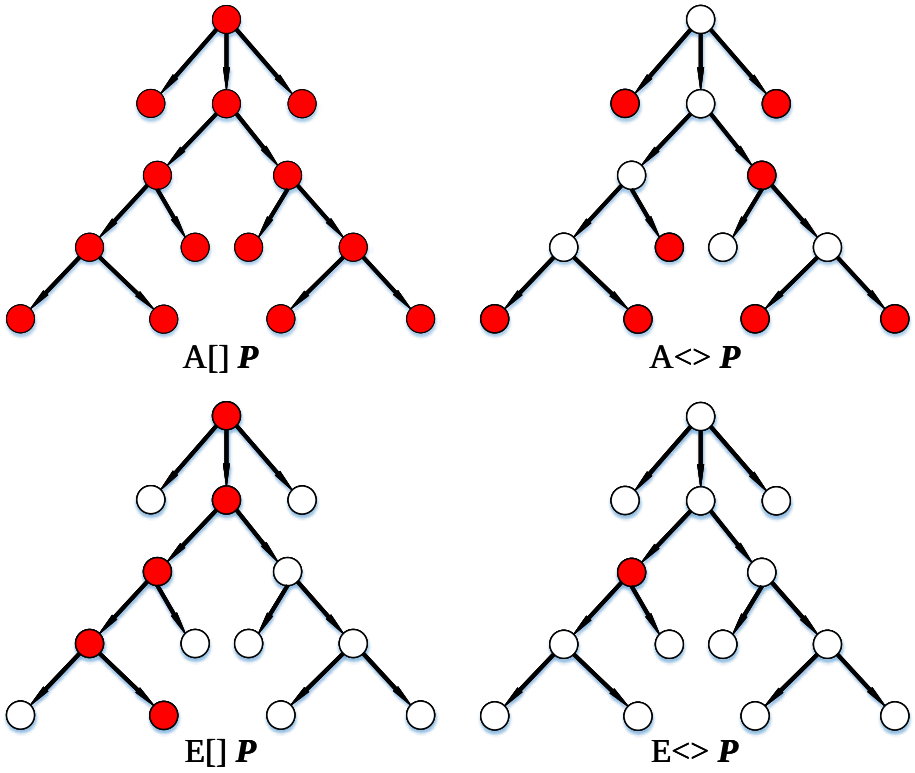


Fig 58 Schematic of the four formulae supported by UPPAAL

As presented in Fig 58, the four formulae of the property descriptions determine the checking scale of the TA model. The formula '*A* [] ***P***' requires that the property ***P*** should be satisfied in all states of all the traces contained in the TA model. The formula '*A* <> ***P***' requires that the property ***P*** should be satisfied in some states of all the traces contained in the TA model. The formula '*E* [] ***P***' requires that the property ***P*** should be satisfied in all the states of some traces contained in the TA model. The formula '*E* <> ***P***' requires that the property ***P*** should be satisfied in some states of some traces contained in the specification model. With the four specific formulae, UPPAAL can automatically check the safety and liveness properties of the

built model, determining whether these properties comply with the system specification. The safety property requires that unexpected events never happen during the system operation process, which means the corresponding formulae are 'A [] $\boldsymbol{P}$' and 'E [] $\boldsymbol{P}$'. A special formula '$\boldsymbol{P_1} \rightarrow \boldsymbol{P_2}$' for safety property verification is supported by UPPAAL, meaning that $\boldsymbol{P_2}$ will be eventually satisfied whenever $\boldsymbol{P_1}$ is satisfied. The liveness property requires that expected events can eventually happen during the system operation process, which means the corresponding formulae are '$A <> \boldsymbol{P}$' and '$E <> \boldsymbol{P}$'.

The purpose of validating the specification model is to confirm that it correctly describes the system specification so that the testing results are always obtained from the correct test oracle. The author validated the specification model by verifying whether it satisfies the safety and liveness properties which are desired by the system specification. Although consistency between the specification model and system specification cannot be completely proven in this way, validation of the specification model can guarantee that no safety or liveness errors exist in the specification model, which can adequately prove that the specification model is eligible to be used in black-box testing. Fig 59 shows an overview of all the verified properties in the specification model:

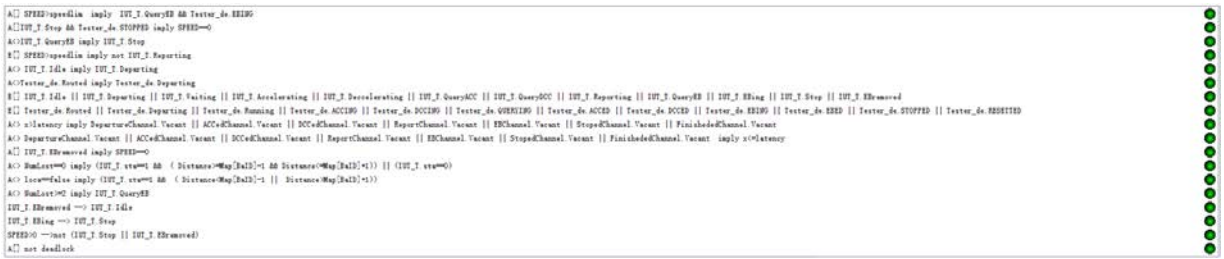


Fig 59 Summary of all verified safety and liveness properties

Based on Fig 59, all the safety and liveness properties pass the verification via the integrated model-checking tool box in UPPAAL. The verification formulae are written in the format of first-order logic with a special grammar required by UPPAAL. For example, the formula 'SPEED $> 0 \rightarrow$ not $(IUT\_T.Stop \,||\, IUT\_T.EBremoved)$' means that the states '$Stop$' and '$EBremoved$' in the TA 'IUT_T' can never be available when 'SPEED > 0' holds, which requires that the SUT VOBC (train) can only be stopped and the EB released when the train speed is zero. The safety property is derived from the test specification with a different angle of description which can be intuitively comprehended by humans and read by computers. With the help of model-checking, the liveness and safety properties were verified, and the verification procedure is presented.

6.1.1.1 Deadlock

The first essential verification which should be implemented is to verify that the TA model built has no deadlock. It is the most important verification because a deadlock in the TA model may lead to inconclusive situations during the testing process, making all the covered situations meaningless. To verify the TA model is deadlock-free, the formula 'A $[]not\ deadlock$' is used in UPPAAL.

6.1.1.2 Safety Properties

In this section, the verified properties are explained in detail, in terms of the meaning of the properties, the reasons for verifying them, and the verification results.

*6.1.1.2.1 A[] **SPEED > speedlim** imply IUT_T.**QueryEB** && Tester_de.**EBING***

- Meaning: in all states in all the traces contained in the TA model, the condition '**SPEED > speedlim**' being true implies that the TA models of '*IUT_T*' and '*Tester_de*' will eventually turn into the EB mode, which is presented by the states '***QueryEB***' and '***EBING***'.
- Reason: the key function of overspeed protection is to protect the train from overspeeding, by decelerating the train when the train speed is too fast. This property aims to verify whether the TA model goes into EB mode when the train is overspeeding because the EB function is only available in EB mode.

*6.1.1.2.2 A[]IUT_T.**Stop** && Tester_de.**STOPPED** imply **SPEED** == 0*

- Meaning: when the TA model '*IUT_T*' is in the state '***Stop***', and the TA model '*Tester_de*' is in the state '***STOPPED***', the train speed '**SPEED**' is implied to be zero.
- Reason: as required by the specification, the SUT train should eventually be stopped once the EB is triggered. The states '***Stop***' and '***STOPPED***' stand for the stopped states in the SUT and the tester, where the train speed '**SPEED**' should always be zero. This property verifies that the SUT can achieve the stopped state only when the train speed is down to zero.

*6.1.1.2.3 A[] **SPEED > speedlim** imply not IUT_T.**Reporting***

- Meaning: when the train speed '**SPEED**' is greater than the speed limit '**speedlim**', the state '***Reporting***' which stands for the SUT being in the normal operation mode becomes unavailable.

- Reason: this property aims to check that the SUT cannot stay in the normal operation mode when the train speed '**SPEED**' exceeds the speed limit '**speedlim**', which means that the SUT should enter EB implementation mode when overspeed happens. Satisfaction of the property guarantees that the SUT must go to EB mode when an overspeed situation is detected.

6.1.1.2.4 $A[]\ IUT\_T.\boldsymbol{EBremoved}\ imply\ \boldsymbol{SPEED} == 0$

- Meaning: when the SUT is in the state '$\boldsymbol{EBremoved}$', the train speed '**SPEED**' is always zero.
- Reason: another safety-critical function related to overspeed protection is that the implemented EB can be released only when the train is completely stopped. State '$\boldsymbol{EBremoved}$' stands for the condition where the implemented EB has been released from the SUT train, and the event can happen only when the train speed '**SPEED**' is zero. Satisfaction of the property guarantees that removal of the EB happens in safe conditions.

6.1.1.2.5 $E[]\ \boldsymbol{loca} == false\ imply\ (IUT\_T.\boldsymbol{sta} == 1\ \&\&\ (\boldsymbol{Distance} < Map[BaID] - 1\ \vee\ \boldsymbol{Distance} > Map[BaID] + 1))$

- Meaning: the Boolean variable '**loca**' being false implies that the variable '**sta**' equals 1, and the variable '**Distance**' is out of the valid receiving range of a certain balise.
- Reason: one reason that train location is missed happens because the VOBC receives a valid balise ID without a valid balise central position. The Boolean variable '**loca**' being false represents that the train position is lost, and the variable '**sta**' means that the SUT is receiving

a valid balise ID. Satisfaction of the property guarantees that the SUT can reject a received balise ID when its corresponding central position is wrong. The reason for using the formula '$E[]$' but not '$A[]$' is that loss of train location happens in several situations, and the one presented by the property is only one of them.

*6.1.1.2.6 A[] **NumLost** >= 2 imply IUT_T.**QueryEB***

- Meaning: when the variable '**NumLost**' is no less than 2, the SUT always goes to the state '***QueryEB***' eventually.
- Reason: when two continuous balises are found to be missed, the variable '**NumLost**' becomes 2, and the SUT should find that the train location is missed. In this situation, no matter what current state the SUT is in, it should trigger the EB immediately and stop the train eventually. Satisfaction of the property indicates that the SUT can detect that the train position is lost and go to the EB mode to keep the train safe.

*6.1.1.2.7 IUT_T.**EBremoved** -> IUT_T.**Idle***

- Meaning: the state '***EBremoved***' in the TA model '*IUT_T*' always leads to the state '***Idle***'.
- Reason: when the EB is removed from the SUT train, the SUT should eventually be able to go to the initial state. Satisfaction of the property guarantees that the SUT TA model does not have deadlock in the state '***EBremoved***'.

*6.1.1.2.8 IUT_T.**EBing** -> IUT_T.**Stop***

- Meaning: the state '***EBing***' in the TA model '*IUT_T*' always leads to the state '***Stop***'.

- Reason: the SUT will eventually go to the state '***Stop***' if its current state is '***EBing***', which indicates that the train should eventually be stopped once the EB is triggered. Satisfaction of the property guarantees that the EB is effective in stopping the train.

6.1.1.2.9 $\boldsymbol{SPEED} > 0 \text{ -> } not\ (IUT\_T.\boldsymbol{Stop} \vee IUT\_T.\boldsymbol{EBremoved})$

- Meaning: the variable '**SPEED**' being greater than zero leads to the SUT not being able to go to the state '***Stop***' or '***EBremoved***'.
- Reason: the SUT should never go to the state '***Stop***' or '***EBremoved***' before the train is completely stopped. As a result, satisfaction of the property guarantees that the SUT stays in the EB mode when the train speed is not zero.

6.1.1.3 Liveness Properties

6.1.1.3.1 $A <> IUT\_T.\boldsymbol{QueryEB}\ imply\ IUT\_T.\boldsymbol{Stop}$

- Meaning: when the SUT is in the state '***QueryEB***', it will eventually go to the state '***Stop***'.
- Reason: satisfaction of the property indicates that the SUT can be completely stopped by the EB, which means that the expected state '***Stop***' can be achieved eventually.

6.1.1.3.2 $A <> IUT\_T.\boldsymbol{Idle}\ imply\ IUT\_T.\boldsymbol{Departing}$

- Meaning: when the SUT is in the state '***Idle***', it will eventually go to the state '***Departing***'.
- Reason: the testing purpose requires that the train can departure eventually. Satisfaction of the property guarantees that the train will not always be stuck in the initial state '***Idle***' and will eventually depart at some time.

*6.1.1.3.3 A <> Tester_de.**Routed** imply Tester_de.**Departing***

- Meaning: when the tester is in the state '***Routed***, it will eventually go to the state '***Departing***'.
- Reason: the testing purpose requires that the tester should send the departure command to the SUT at some time. Satisfaction of the property indicates that the tester will try to send the departure command to the SUT train and make the following testing steps available.

6.1.1.3.4 A <>

$$IUT\_T.\mathbf{Idle} \vee IUT\_T.\mathbf{Departing} \vee IUT\_T.\mathbf{Waiting} \vee IUT\_T.\mathbf{Accelerating} \vee IUT\_T.\mathbf{Deccelerating} \vee IUT\_T.\mathbf{QueryACC} \vee IUT\_T.\mathbf{QueryDCC} \vee IUT\_T.\mathbf{Reporting} \vee IUT\_T.\mathbf{QueryEB} \vee IUT\_T.\mathbf{EBing} \vee IUT\_T.\mathbf{Stop} \vee IUT\_T.\mathbf{EBremoved}$$

- Meaning: all the states contained in the SUT TA model should be reachable in some traces.
- Reason: satisfaction of the property indicates that there is no unreachable state in the SUT model so that everything defined in the model can be covered in the testing process at some time.

6.1.1.3.5 $A <> Tester\_de.\mathbf{Routed} \vee Tester\_de.\mathbf{Departing} \vee Tester\_de.\mathbf{Running} \vee Tester\_de.\mathbf{ACCING} \vee Tester\_de.\mathbf{DCCING} \vee Tester\_de.\mathbf{QUERYING} \vee Tester\_de.\mathbf{ACCED} \vee Tester\_de.\mathbf{DCCED} \vee Tester\_de.\mathbf{EBING} \vee Tester\_de.\mathbf{EBED} \vee Tester\_de.\mathbf{STOPPED} \vee Tester\_de.\mathbf{RESETTED}$

- Meaning: all the states contained in the tester TA model should be reachable in some traces.
- Reason: satisfaction of the property indicates that there is no unreachable state in the tester model so that everything defined in the model can be covered in the testing process at some time.

6.1.1.3.6 $A <> \boldsymbol{x} >$ $latency\ imply\ DepartureChannel.\mathbf{Vacant} \vee ACCedChannel.\mathbf{Vacant}\ DCCedChannel.\mathbf{Vacant} \vee ReportChannel.\mathbf{Vacant} \vee EBChannel.\mathbf{Vacant} \vee StopedChannel.\mathbf{Vacant} \vee FinishededChannel.\mathbf{Vacant}$

- Meaning: when the clock '$\boldsymbol{x}$' is larger than the '$latency$', one of the communication channels must be in the state '$\mathbf{Vacant}$'.
- Reason: satisfaction of the property indicates that the clock can only be greater than the latency when in the state '$\mathbf{Vacant}$'.

6.1.1.3.7 $A <> DepartureChannel.\boldsymbol{Busy} \vee ACCedChannel.\boldsymbol{Busy} \vee DCCedChannel.\boldsymbol{Busy} \vee ReportChannel.\boldsymbol{Busy} \vee EBChannel.\boldsymbol{Busy} \vee StopedChannel.\boldsymbol{Busy} \vee FinishededChannel.\boldsymbol{Busy}\ imply\ \boldsymbol{x} <= latency$

- Meaning: when the communication channels are in the state '$\boldsymbol{Busy}$', the clock '$\boldsymbol{x}$' must be no greater than the '$latency$'.
- Reason: satisfaction of the property indicates that the state '$\boldsymbol{Busy}$' is only available when the clock '$\boldsymbol{x}$' is within the '$latency$', which means that all the communication channels must go from the state '$\boldsymbol{Busy}$' to the state '$\boldsymbol{Vacant}$' once the clock exceeds the '$latency$'.

6.1.1.3.8 $A <> \boldsymbol{NumLost} == 0\ imply\ (IUT\_T.\boldsymbol{sta} == 1\ \&\&\ (\boldsymbol{Distance} >= Map[BaID] - 1\ \&\&\ \boldsymbol{Distance} <= Map[BaID] + 1))\ ||\ (IUT\_T.\boldsymbol{sta} == 0)$

- Meaning: the variable '**NumLost**' is equal to zero when the SUT receives a valid balise number with a valid balise central position, or the SUT does not come across balises.
- Reason: when the SUT is running normally on the track without overspeed, it can receive a valid balise ID with a correct balise central position, or it can run without receiving balises. Satisfaction of the property indicates that the train location function of the SUT VOBC performs correctly according to the specification.

6.1.2 Simulation Model Validation

Unlike verification of the abstract model which is in TA format, the simulation model of the specification model cannot take advantage of the model-checking integrated into UPPAAL,

which makes its formal verification more expensive than that of the abstract model. The simulation models developed in this thesis is assumed to be correct and comply with the system specification to undertake the key testing tasks. In engineering practice, the simulation models could be either developed according to the system specification or adopted directly from the system software with addition of simulation control models.

6.2 Effectiveness Verification

To prove that the simulation combined MBT performs is better than existing testing approaches, effectiveness verification and performance verification are undertaken to prove that the developed testing platform can detect errors and achieve a better coverage. If the developed testing platform is evaluated to detect every error covered and to cover more possibilities, it is a better testing approach because it has a higher possibility to find error hidden in the SUT than existing ones.

6.2.1 Mutation Testing

With the specification model verified, the effectiveness of simulation combined MBT can be verified to determine whether the testing platform can find out errors in an SUT. Since the testing results in Chapter 5 indicate that there are no errors in the SUT VOBC, the author verified the testing platform further to see whether it can find existing errors in an SUT mutation, which is obtained by injecting known errors into the SUT. The verification process is called mutation testing in the computer science field and contains a set of different kinds of mutation operators corresponding to different errors [127, 128]. The application domain for

this thesis is the rail industry and as a result, the author has simplified the mutation testing by only selecting mutation operators which are meaningful in railway system testing. Table 9 shows the summary of a set of mutation testing results:

Mutation error	Errors inserted into the SUT	Test results
Wrong output action	e.g. make the SUT send out '*ACCed*' when it receives '*DCC*' from the testing platform	Passed
Incorrect output value	e.g. make the train speed decelerate with the output action '*ACCed*' when the SUT receives '*ACC*'	Passed
Delay	e.g. insert a major delay of 200 ms in the communication channel	Passed
Missing state	e.g. remove the state '***Accelerating*** (see Fig 47) in the SUT	Passed
Transition to wrong state	e.g. change the transition '***Reporting***' to "***QueryEB***" (see Fig 47)	Passed
Incorrect initial condition	e.g. give the SUT a wrong initial state	Passed

Table 9 Summary of mutation testing results

As shown in Table 9, six mutation tests were implemented to verify whether the simulation combined MBT platform can detect known errors. All the mutation testing presented typical errors which can be found in black-box testing, and the verification results indicate that the

SUT VOBC can detect all the errors inserted. The details of the six mutation testing are presented as follows:

6.2.1.1 Wrong Output Action

It is the most basic function that a testing platform should detect unexpected output actions. The author inserted the error by modifying the SUT code, making the SUT send out '*ACCed*' when it receives a '*DCC*' command. Therefore, the mutated SUT accelerates a train when it receives a decelerating command, which does not satisfy the specification. The verdict given by the testing platform indicates that the error was detected.

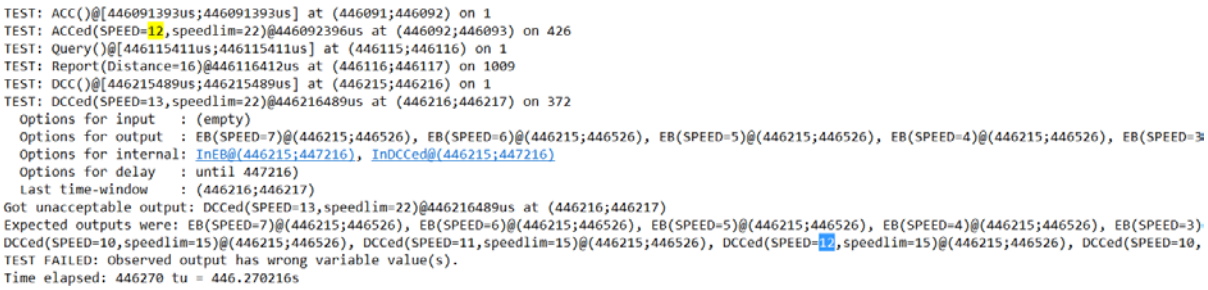
```
TEST: DCC()@[6958153us;6958153us] at (6958;6959) on 1
TEST: ACCed(SPEED=3,speedlim=22)@6960134us at (6960;6961) on 204
  Options for input   : (empty)
  Options for output  : EB(SPEED=22)@(6958;7269), EB(SPEED=23)@(6958;7269), DCCed(SPEED=3,speedlim=4)@(6958;7269), DCCed(SPEED=2,speedlim=4)@(6958;7269)
  Options for internal: InEB@(6958;7959), InDCCed@(6958;7959)
  Options for delay   : until 7959)
  Last time-window    : (6960;6961)
Got unacceptable output: ACCed(SPEED=3,speedlim=22)@6960134us at (6960;6961)
Expected outputs were: EB(SPEED=22)@(6958;7269), EB(SPEED=23)@(6958;7269), DCCed(SPEED=3,speedlim=4)@(6958;7269), DCCed(SPEED=2,speedlim=4)@(6958;7269),
TEST FAILED: Observed unacceptable output.
Time elapsed: 6989 tu = 6.989156s
Time    left: 9993011 tu = 9993.010844s
Random  seed: 1495188639
```

As indicated by the verdict, the expected output action corresponding to the input command '*DCC*' is 'EB' or '*DCCed*' while the received output action is '*ACCed*'. Therefore, the testing platform drew a failed conclusion and interrupted the testing process.

6.2.1.2 Incorrect Output Value

Another basic function for testing a platform in black-box testing is to check whether the output variable value is correct according to the testing specification. A wrong variable value along with a correct output action should be discovered. The author inserted the error by making the SUT train brake when it receives the input command '*ACC*' and feeds back the output action '*ACCed*'. Although the input and output actions comply with the expected ones,

the testing platform should still find inconsistencies in train speed between the SUT and the specification. The verdict given by the testing platform indicates that the error was detected.



As indicated by the verdict, the expected output value of the variable '**SPEED**' should be no more than 12 according to the previous value of '**SPEED**' along with the previous output action '*ACCed*'. The testing platform detected the error and drew a failed conclusion for the testing.

6.2.1.3 Delay

For testing of the SUT containing time constraints, the testing platform is required to determine whether delays between input and output actions comply with the specification requirement. In black-box testing, an output should arrive in time after the input action, which means a delayed output action should draw a failed conclusion. The author inserted the error by adding a response delay of 1000 time-units between the input action '*ACC*' and its corresponding output action '*ACCed*', which makes the '*ACCed*' arrive later than the time constraints in the specification. The verdict given by the testing platform indicates that the error was detected.

```
TEST: ACC()@[3018592us;3018592us] at (3018;3019) on 1
TEST: delay to (4019 on 138
  Options for input   : (empty)
  Options for output  : ACCed(SPEED=0,speedlim=15)@(4017;4019), ACCed(SPEED=2,speedlim=14)@(4017;4019), ACCed(SPEED=1,speedlim=14)@(4017;4019),
  Options for internal: InACCed@(4017;4019), InFB@(4017;4019)
  Options for delay   : until 4019)
  Last time-window    : (4019;4020)
  Max. system delay   : until 4019)
Last time-window is beyond maximum allowed delay.
TEST FAILED: IUT failed to produce output in time.
Time elapsed: 4036 tu = 4.036109s
Time    left: 9995964 tu = 9995.963891s
Random  seed: 1495198016
```

As indicated by the verdict, the testing platform expects the output '*ACCed*' to arrive with 1000 time-units after the input action '*ACC*' happens. The time stamp of the '*ACC*' is 3018, and the testing platform did not receive the expected output action '*ACCed*' until the time stamp went to 4019. Since overtime happened between the input and output actions, the testing platform detected the inconsistency and drew a failed conclusion.

6.2.1.4 Missing State

A missing state can happen when synchronisation of the SUT and testing platform is broken, making the SUT transition miss a certain state and jump over to a further one. It is important for a testing platform to detect this unusual situation in the implementation of black-box testing. Broken synchronisation should terminate the testing process immediately because the following testing results are all based on wrong synchronisation. The author inserted the error by making the SUT skip the state '***Accelerating***' to arrive at state '***QueryACC***' directly. The verdict given by the testing platform indicates that the error was detected.

```
TEST: Depart()@[3001998us;3001998us] at (3001;3002) on 1
TEST: delay to (3002 on 2
TEST: delay to [3004 on 3
TEST: Departed()@3005000us at [3005;3005] on 4
TEST: ACC()@[3017078us;3017078us] at (3017;3018) on 1
TEST: Report(Distance=0)@3018008us at (3018;3019) on 138
Short post-mortem analysis based on last good stateSet(138):
Got unacceptable output: Report(Distance=0)@3018008us at (3018;3019)
Expected outputs were: ACCed(SPEED=0,speedlim=15)@(3017;3328), ACCed(SPEED=2,speedlim=14)@(3017;3328), ACCed(SPEED=1,speedlim=14)@(3017;3328),
TEST FAILED: Observed unacceptable output.
Time elapsed: 3044 tu = 3.044045s
Time    left: 9996956 tu = 9996.955955s
Random  seed: 1495200071
```

As indicated by the verdict, the missing state was found by the testing platform because the

testing platform received an unacceptable output action ‘Report’, missing the correct one which is ‘*ACCed*’. The reason this happens is that the output action ‘*ACCed*’ becomes invalid in any other state except for ‘***Accelerating***’. Therefore, the testing platform detected the inconsistency and drew a failed conclusion.

6.2.1.5 Transition to Wrong State

The error of the SUT transiting to a wrong state can happen when the internal logic of the SUT is falsified. Transition to a wrong state makes the following input and output actions conflict with the expected pattern. Therefore, the testing platform should detect this error in black-box testing. The author inserted the error by falsifying the state transition logic of the SUT, making it transit from the state ‘***Reporting***’ to ‘***QueryEB***’ no matter which output action is available. The verdict given by the testing platform indicates that the error was detected.

```
TEST: Departed()@3002355us at (3002;3003) on 4
TEST: ACC()@[3012365us;3012365us] at (3012;3013) on 1
TEST: ACCed(SPEED=0,speedlim=22)@3013349us at (3013;3014) on 138
TEST: Query()@[3035369us;3035369us] at (3035;3036) on 1
TEST: Report(Distance=0)@3036372us at (3036;3037) on 1024
TEST: ACC()@[3137455us;3137455us] at (3137;3138) on 1
TEST: EB(SPEED=0)@3138457us at (3138;3139) on 138
TEST: Query()@[3153487us;3153487us] at (3153;3154) on 1
TEST: Stoped(SPEED=0,Des=0)@3154468us at (3154;3155) on 27
TEST: Unlock()@[13154797us;13156300us] at (13154;13157) on 1
TEST: Finished()@13155798us at (13155;13156) on 2
TEST: Depart()@[16155436us;16157437us] at (16155;16158) on 1
TEST: Departed()@16157437us at (16157;16158) on 2
TEST: ACC()@[16169945us;16169945us] at (16169;16170) on 1
TEST: ACCed(SPEED=10,speedlim=22)@16170951us at (16170;16171) on 138
Short post-mortem analysis based on last good stateSet(138):
  Options for input   : (empty)
  Options for output  : ACCed(SPEED=0,speedlim=15)@(16169;16480), ACCed(SPEED=2,speedlim=14)@(16169;16480), ACCed(SPEED=1,speedlim=14)@(16169;16480),
  Options for internal: InACCed@(16169;17170), InEB@(16169;17170)
  Options for delay   : until 17170)
  Last time-window    : (16170;16171)
Got unacceptable output: ACCed(SPEED=10,speedlim=22)@16170951us at (16170;16171)
Expected outputs were: ACCed(SPEED=0,speedlim=15)@(16169;16480), ACCed(SPEED=2,speedlim=14)@(16169;16480), ACCed(SPEED=1,speedlim=14)@(16169;16480),
TEST FAILED: Observed output has wrong variable value(s).
Time elapsed: 16228 tu = 16.228991s
Time    left: 9983772 tu = 9983.771009s
Random  seed: 1495201554
```

As indicated by the verdict, the inserted error was found by the testing platform. Since the transition logic was falsified, the SUT was forced to go to state ‘***QueryEB***’, and make a series of wrong input and output actions. After the SUT went back to the correct state, the testing

platform detected the inconsistency in the variable '**SPEED**' and drew a failed conclusion for the testing.

6.2.1.6 Incorrect Initial state

The last error is caused by incorrect initialisation of the SUT, which makes all the following testing results meaningless. Therefore, the testing platform should be able to find that the SUT is incorrectly initialised at the beginning of the testing process. The author inserted the error by giving the SUT a wrong initial state, '***Reporting***'. The verdict given by the testing platform indicates that the error was detected.

```
TEST: Depart()@[3001791us;3001791us] at (3001;3002) on 1
  Options for input   : (empty)
  Options for output  : Departed()@(4001;4002)
  Options for internal: InDeparted@(4001;4002)
  Options for delay   : until 4002)
  Last time-window    : (4002;4003)
  Max. system delay   : until 4002)
Last time-window is beyond maximum allowed delay.
TEST FAILED: IUT failed to produce output in time.
Time elapsed: 4002 tu = 4.002416s
Time    left: 9995998 tu = 9995.997584s
Random  seed: 1495202225
```

As indicated by the verdict, the testing drew a failed conclusion at the beginning of testing because the first output action '*Departed*' could not be observed by the testing platform. The wrong initial state blocked the SUT from sending out any valid output actions. Therefore, the testing platform detected the inconsistency.

6.2.2 Reachset Conformance Relation

The presented testing results for the mutation testing prove that the system can detect most of the known errors which may lead to dangerous situations of system operation such as

overspeed or lost train location. The soundness of the simulation combined MBT platform can be proven under the assumption that all the errors which potentially exist in the SUT are completely ascertained. However, even for an experienced tester, it is not possible to spot all potential errors that could lead to dangerous situations in the operation of complex systems such as TCSs. To verify that the testing platform can detect unknown errors, the testing results should be analysed to find out whether there is any inconsistency issue between the results obtained and the system specification. However, straightforward verification of the testing result is expensive (time and resource usage). The purpose of testing verification is to prove that the testing does not miss any errors which may become potential risks in the future. As a result, the verification can be transformed to prove that no errors violating the safety properties are missed in the testing process. To achieve that goal, the author applies a concept of conformance relation in the verification, which is less expensive than verifying the trace conformance relation adopted but strong enough to prove that the SUT satisfies the safety properties in the specifications. This relation is called Reachset Conformance Relation and has been defined by Roeahm et al. [129].

Different from the trace conformance relation, the reachset conformance relation determines the conformance relationship between two systems (abstract system and real system) by proving the inclusion relationship of their reachable input set and output space. To explain the verification method using application of the reachset conformance relation, the author compared the traditional trace conformance relation and the reachset conformance relation, illustrated by Fig 64:

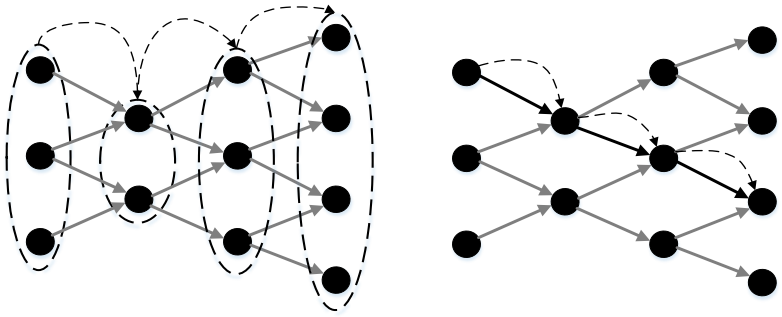


Fig 60 Comparison between reachset conformance and trace conformance

As revealed by Fig 60, different from the trace conformance relation which is applied to construct the testing platform, the reachset conformance relation recombines the states into a group of reachable sets by hiding the individual transitions from a certain state to another. Based on the concept of reachset, the definition of the reachset conformance relation in the field of black-box testing can be formally obtained:

Letting S_{spec} *and* S_{SUT} *be implementation of the specification and the SUT,* S_{SUT} *is reachset-conformant to* S_{spec} *if the input set and output space of* S_{SUT} *are a subset of the input set and output space of* S_{spec}.

Based on Fig 60, the reachset conformance relation holds if the trace conformance relation is satisfied between two implementations. Therefore, the reachset conformance relation is a weaker relation than the trace conformance relation, which means that trace conformance cannot be proven by verifying reachset conformance. However, the purpose of testing platform verification is to prove that the SUT complies with the safety properties desired by the specifications. Trace conformance is one way to achieve that goal. To verify the

conclusion drawn from the trace conformance relation, the reachset conformance relation can be used to check the result from another point of view. For the application of reachset conformance relation, the task is to check whether the safety properties are satisfied by the SUT with the reachset conformance relation, which means the SUT never enters a dangerous area. Since the input set of the SUT for black-box testing is derived from the specification, it is unnecessary to prove that the input set of the SUT is a subset of the input set of the specification, because any invalid input from the specification will be directly rejected by the test tool TRON without sending it to the SUT. Therefore, two main safety properties of the output space should be always satisfied during the system operation procedure:

a. The train speed should never exceed the speed limit by the overspeed tolerance of 5 km/h.

b. Two trains should never be at the same point along the track in the time region.

Since verification of the two safety properties can be solved within the two-dimensional region (one variable versus testing time), the reachset of the system can be obtained directly from the SUT, from the data recorded during testing. By comparing the output reachset obtained from the SUT and the output reachset specified by the safety properties, the reachset conformance relationship between the SUT and the safety properties can be determined, as indicated by Fig 61 and Fig 62:

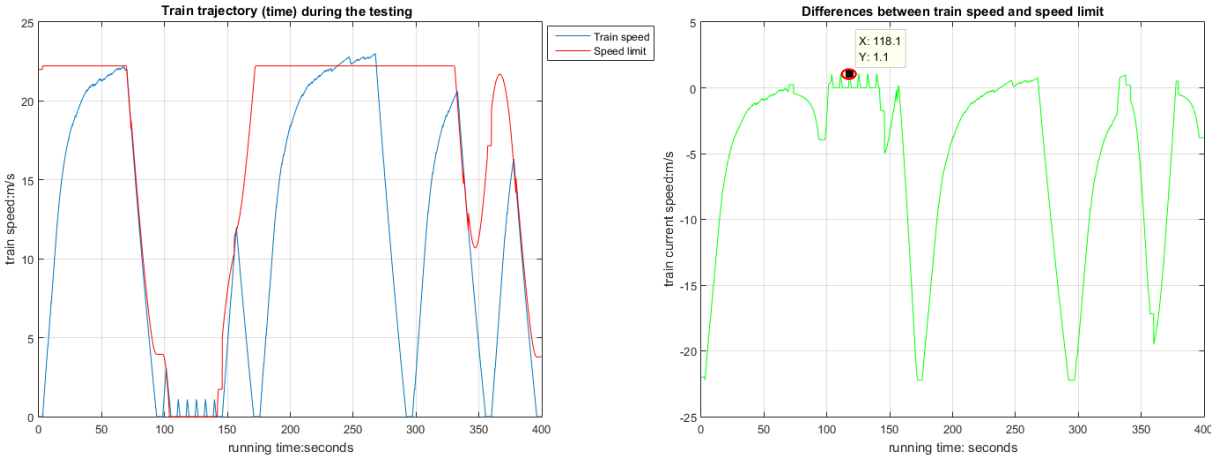


Fig 61 Example of differences between train speed and speed limit

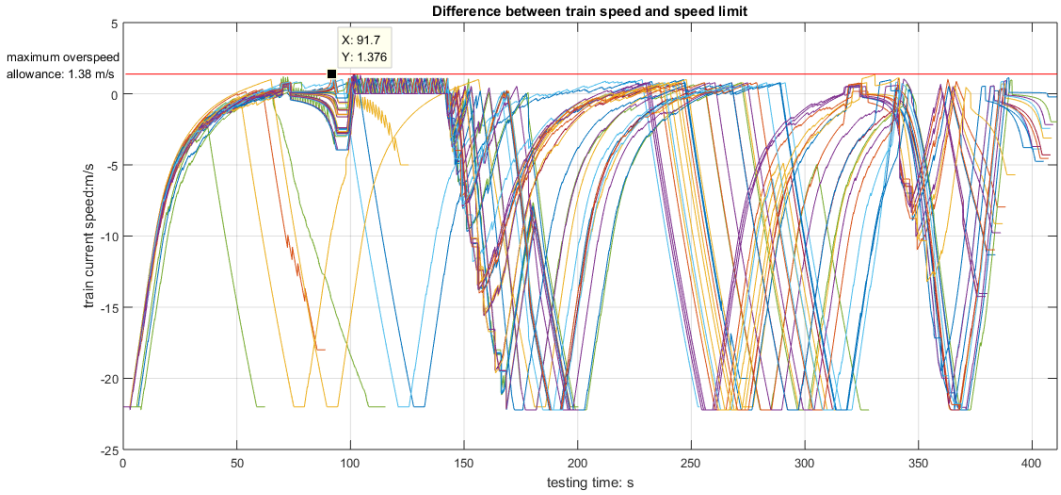


Fig 62 Verification results for the reachset conformance relation

Based on the safety property ‘a’, the reachset of the specification can be obtained as follows:

$$Reachset_{spec} = Train\ Speed\ \epsilon\ [0, 85]\ \left(\frac{km}{h}\right), \forall\ t \in testing\ time$$

According to the right-hand graph in Fig 61, the maximum difference between the train speed and speed limit during testing is obtained as $1.1\ \mathrm{m/s}$, which is below the maximum overspeed allowance of $5\ \mathrm{km/h}$ $(1.38\ \mathrm{m/s})$. In the left-hand graph in Fig 61, the maximum

train speed can be obtained as $23\,\mathrm{m/s}$ $(82.8\,\mathrm{km/h})$, which is below the theoretical maximum speed which can be achieved by the train in the network, $23.6\,\mathrm{m/s}$ $(80 + 5 = 85\,\mathrm{km/h})$. Therefore, according to the testing results shown in Fig 61, the SUT is reachset-conformant to the safety property 'a' in the recorded testing time, which means the SUT behaviour complies with the specification in the testing time. In Fig 62, the same verification method is applied to all the trajectories recorded in the testing. The straight red line represents the threshold which should not be surpassed by any train trajectory. The maximum train speed is obtained as shown in Fig 62, which indicates that the maximum difference between the train trajectory and the speed limit is $1.376\,\mathrm{m/s}$. Therefore, the reachset conformance relation is satisfied between the SUT and the specification, since no counterexample is found.

Similarly, the reachset of the specification for property 'b' can be obtained as below:

$$Reachset_{spec} = Train\ location_{SUT}(t), \text{where } \forall\ t \in testing\ time, Train\ location_{SUT}(t)$$

$$\neq Train\ location_{front}(t) \text{ or } Train\ location_{behind}(t)$$

Based on the reachset obtained from the SUT train location, the reachset conformance relation of the safety property 'b' can be verified by the distance–time graphs, as shown by Fig 63:

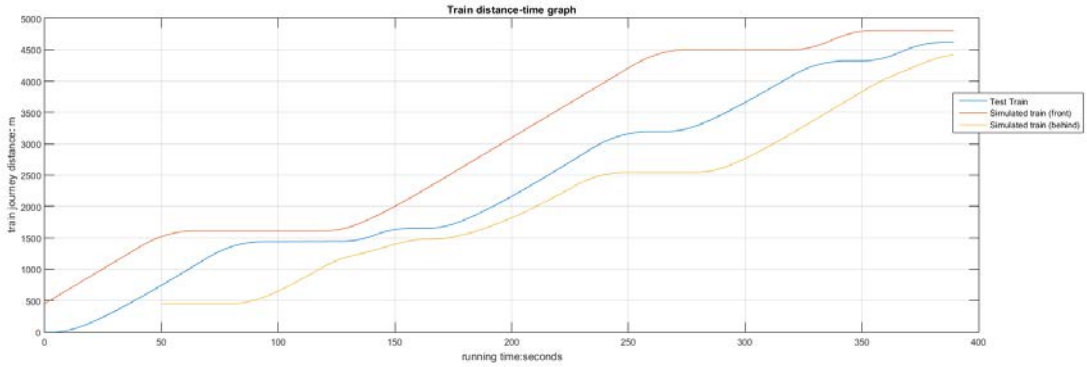


Fig 63 Example of a distance–time graph for verification

As seen from Fig 63 and discussed in Chapter 6, no two trains appear at the same point along the track during testing, which means no collisions happen in the testing procedure. With the collision detection function applied, the safety property ‘b’ is automatically verified after the testing is finished.

The verification results indicate that the SUT behaviour satisfies the specification requirements during testing. With both the trace conformance relation and reachset conformance relation satisfied, the conformance relationship between the SUT and the specification is dually proven. The reachset conformance for verification is a simplified application which only contains two-dimensional issues. As a result, the reachset of the SUT and the specification can be easily obtained without any further process. When there are more than two reachset dimensions, the reachset cannot be directly analysed before being approximated into a two-dimension issue, which makes reachset verification a far more complex verification method. As a result, a precondition of verification with reachset conformance is that the object under verification can be transformed into a set of

two-dimensional sub-objects.

Compared with mutation testing which verifies whether the testing platform can detect errors of known type, reachset conformance verification aims to verify that no errors exist in the testing results so that no errors are missed by the testing platform, regardless of whether the types of errors are known or unknown. Since a correct testing result should contain no inconsistency with the specification, the testing platform can be proven to be effective if no counterexamples can be found in its testing results. With verification of the testing platform in the fields of known errors and unknown errors, the simulation combined MBT platform is proven to be capable of finding most of the significant errors in the SUT. However, all the verification is based on the recorded results, which means an error could still be missed if it is not covered by the testing platform. Therefore, the performance of the testing platform should be verified to find out its coverage ability.

6.3 Performance Verification

Coverage performance of the simulation combined MBT platform can directly influences its ability to find errors. With a low degree of coverage, the testing platform can miss a lot of errors which could be detected if the error situations are achieved. There are series of factors causing poor coverage in MBT testing, including inappropriate modelling, too large a model size, limited testing scenarios, inefficient test generation algorithm, etc. Compared with traditional manual testing, MBT can achieve more extensive coverage since the test generation process is automated with the help of a computer. However, existing offline test

case generation has limitations when coming across complex SUTs and testing scenarios, such as CBTC system testing. In this section, the author analyses the coverage of the simulation combined MBT platform by comparing it with the coverage of traditional offline testing. Different types of coverage are introduced to prove that the testing platform can comprehensively achieve better coverage than existing methods.

6.3.1 Trace Coverage and Variable Coverage

Coverage was originally a concept of offline MBT testing, measuring how many possibilities out of all valid possibilities have been covered by the implemented test. For offline MBT, coverage is obtained by generating test cases from the built specification model, without considering implementation of the generated test cases, which leads to two main limitations. Firstly, the SUT has to be deterministic without interacting with the testing environment, which makes the specification model too complex to achieve good coverage. Secondly, since the whole transition pattern needs to be recorded to calculate the coverage, coverage of offline testing can be limited by the size of the computer memory. In this situation, the computer memory can be exhaustively occupied when the model has a high degree of complexity. Therefore, there are two main factors influencing the coverage of offline MBT, abstract model complexity and search depth. Search depth indicates how far the test generation algorithm has reached to cover the possibilities, where one step means one transition from one location to another. Since a complex model contains a larger possibility space, it can take more steps to achieve equivalent coverage than a simple model, which takes up more computer memory. Even worse, to achieve better coverage, the computer memory cannot be adequate for offline

test generation.

To determine the performance of the developed simulation combined MBT platform, the author compared the coverage measured from the testing results with the coverage derived from offline test generation under the specification model and testing scenarios. Offline testing coverage analysis was realised by a tool box integrated in UPPAAL, Yggdrasil, which applies the test selection criterion of all-transition coverage to generate a set of test cases within a desired search depth [130]. Given a TA model and a certain search depth, the tool box can calculate the number of accessible transitions and available variable values contained in the model and can record how many of them are covered by the generated test cases.

Online testing coverage is obtained by analysing the recorded log file during the testing. Since all possible transitions and variable values have been obtained by offline test generation tool, coverage can be calculated by counting how many of them have been covered by online testing, which is realised by a MATLAB script searching for keywords which stand for transitions and variable values.

To compare the coverage performances of online and offline testing, each coverage in the same search depth are recorded. Based on the graphs of coverage against search depth, the quantised performance comparison between the simulation combined MBT platform and the offline test generation tool can be obtained, as shown in Fig 64 and Fig 65.

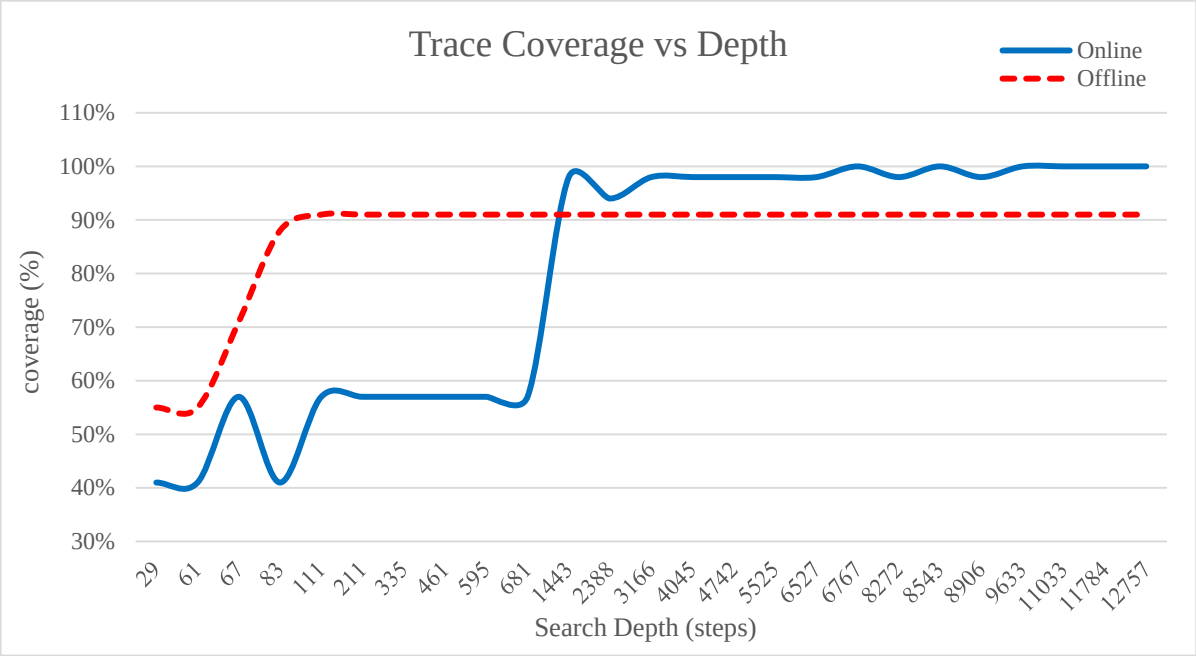


Fig 64 Trace of coverage tendency with search depth

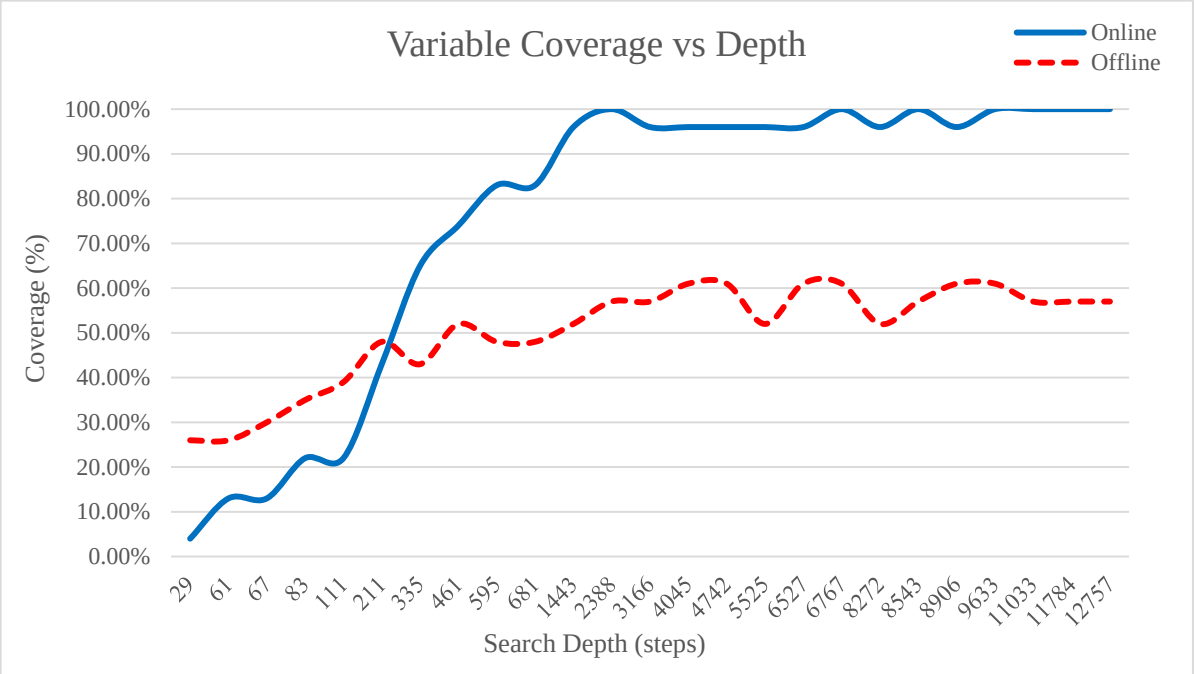


Fig 65 Variable coverage tendency with search depth

As revealed by Fig 64 and Fig 65, the coverage achieved by the simulation combined

platform for the two types of coverage is lower than that of the offline test generation tool when the search depth is low. However, when the search depth increases to 1443 in trace coverage and 211 steps in variable coverage, the coverage performance of the simulation combined MBT platform eventually surpasses that of the offline test generation tool. The reason is that the online test algorithm does not take up an increasing amount of computer memory when the search depth increases, while the offline test algorithm occupies more and more memory along with increasing search depth. Therefore, the offline test algorithm cannot search as deeply as the online test algorithm, as the information it is necessary to record can easily exceed the maximum computer memory with a complex model, which leads to coverage limitations. In the simulation combined MBT platform, abstract model size is extremely reduced by combining modelling with simulation. Furthermore, the online test algorithm simultaneously generates and executes inputs and verdicts for the obtained outputs without recording the information necessary for coverage. As a result, coverage of 100% can eventually be achieved with adequate testing time on the simulation combined MBT platform. However, since the online test generation algorithm randomly selects the valid input based on the current conditions, it cannot positively guarantee or optimise the coverage of test generation. As a result, the author included the simulation model to describe the SUT behaviour more specifically without expanding the size of the TA model, which to some extent strengthens the performance of the original online test generation algorithm, TRON.

6.3.2 Reachset Coverage in Key States

The trace and variable coverage presented is a standard coverage concept originating from

model-checking of the TA model, which can roughly describe the performance of online testing. However, online testing can contain many more possibilities than offline testing due to nondeterminism, which makes trace and variable coverage ineligible to evaluate its performance. The original coverage measures the coverage of the trace and variable values separately, which cannot prove that the whole possibility space is covered. For example, in the overspeed protection function, the two key parameters are train speed and the speed limit. To cover all possibilities, possible combinations of all values of train speed and speed limit should be checked. However, the current variable coverage still individually checks the coverage of the two variables, which misses a lot of potential possibilities contained in the TA model. Therefore, the author introduces a new type of coverage which considers the combination of two key parameters to evaluate variable coverage of the testing platform performance, which is named the reachset coverage in key states.

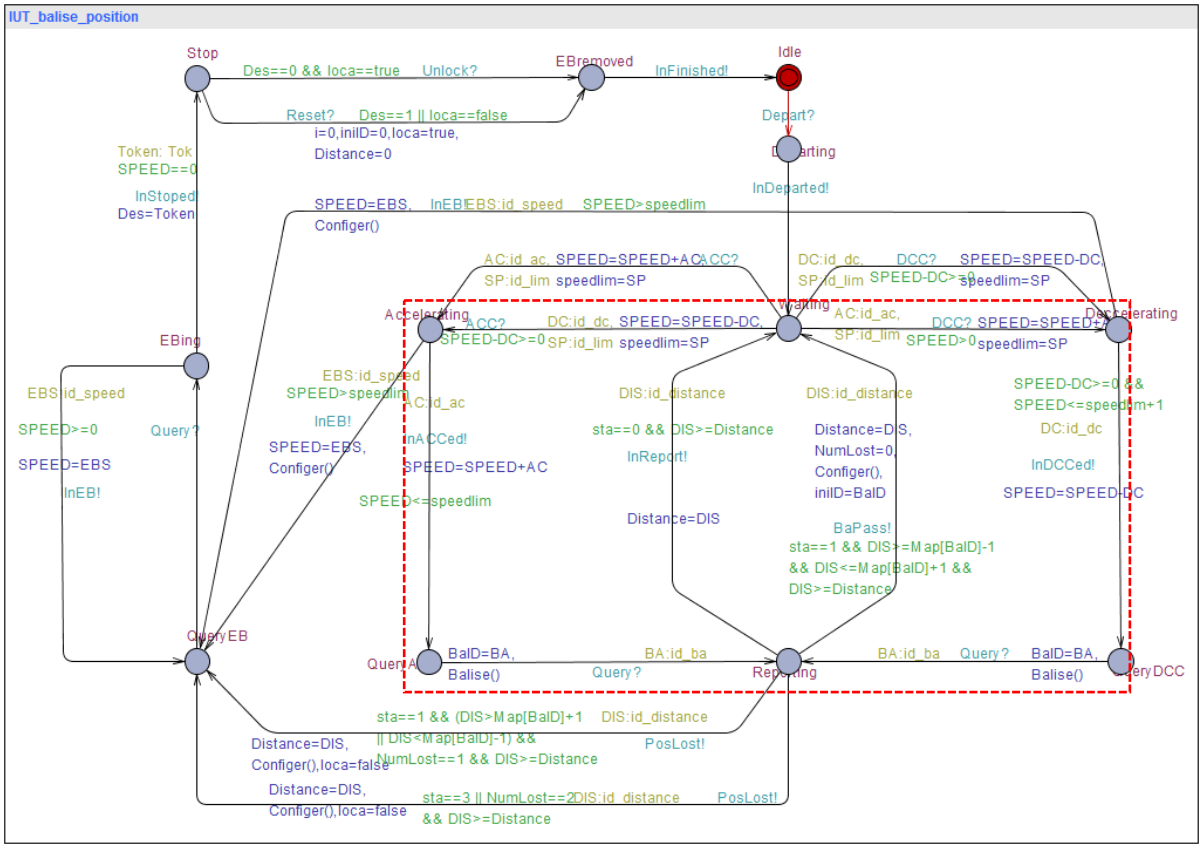


Fig 66 TA model of the SUT in multiple-train scenario

As shown in Fig 66, in the circulation marked in the red dotted box, the values for train speed and speed limit are checked at the same time once for every single loop. Since the kernel function of the overspeed protection is to make different decisions based on the relationship between train speed and speed limit, covering all possible combinations of train speed and speed limit is an essential step to achieve the full coverage of test generation, which means the next reachable set of states from the state '***Waiting***' can be used to determine the variable coverage performance of the overspeed protection. If the testing is passed and covers all possible combinations, the SUT VOBC is proven to be able to always make the correct decision for various speed limits against different train speeds, which means that the

overspeed protection of the SUT VOBC is completely error-free in the given testing environment.

According to the TA model of the SUT VOBC which is presented in Fig 66, the valid value ranges of the speed limit and train speed are both $[0,22]$ m/s. It should be noted that once variable '**SPEED**=23' holds, the TA model breaks out of the circulation in the red box so that '**SPEED**=23' is removed from the reachable set when calculating the coverage, although it is reachable from the state '***Waiting***' in reality. As a result, the coverage matrix can be obtained as a 23×23 matrix which stands for all possible combinations of train speed and speed limit. The reachset coverage calculation method is to search for all the combinations recorded in the testing result and calculate the percentage of the whole coverage matrix covered. The verification results are presented in Fig 67:

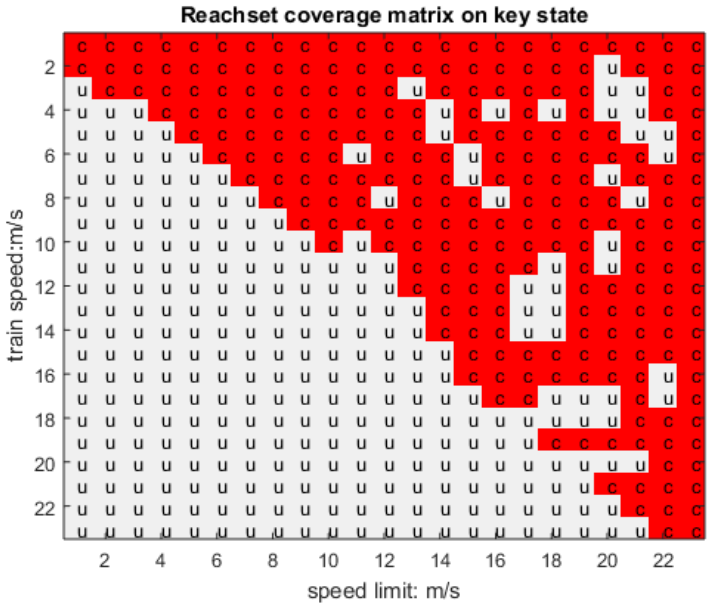


Fig 67 Coverage matrix of testing platform run for 5000 seconds

As shown in Fig 67, the reachset coverage calculated is presented in a 23×23 matrix, where both axes go from 1 to 23, responding to $[0,22]$ m/s, respectively (since the MATLAB index always starts from 1 not 0). The X-axis stands for the speed limit, and the Y-axis stands for the train speed. The yellow area indicates the combinations of the train speed and the speed limit which are covered in the testing according to the log file. Based on the result shown in Fig 67, the reachset coverage in the key states seems to perform poorly during testing, covering less than 50% of possible combinations. The reason is that the actual valid reachset in the key state '***Waiting***' is not the whole 23×23 matrix, which is explained in the following figure:

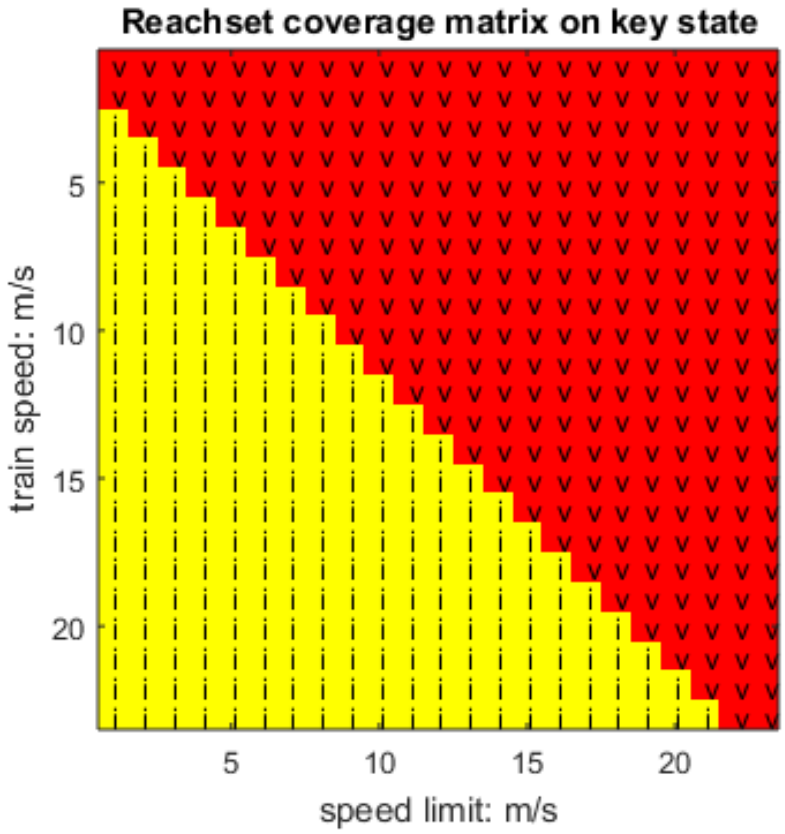


Fig 68 Maximum number of valid combinations of train speed and speed limit

According to the verification results with reachset conformance relation in Section 6.2.2, the valid combinations of the train speed and speed limit should be recalculated, as illustrated by Fig 68. Since the SUT VOBC triggers the EB when the train is overspeeding, the maximum speed which can be achieved by the train is $(V_{lim} + 1)\ \mathrm{m/s}$ (considering the communication delays and overspeed allowance mentioned in the case study), where V_{lim} stands for the current speed limit when the train goes to overspeed. As a result, the expected valid area of the combination of train speed and speed limit should be marked as the red area in Fig 68, which means that the train speed can be $1\ \mathrm{m/s}$ faster than the speed limit. Therefore, the number of valid combinations can be calculated by using the equation:

$$Num_{Valid} = R^2 - \frac{(1 + R - 2) \times (R - 2)}{2} = \frac{R^2 + 3R - 2}{2}$$

where Num_{Valid} is the number of valid combinations of train speed and speed limit, and R stands for the number of the matrix index and satisfies $R \in \boldsymbol{N}$ and $R \geq 1$. Letting R equal $2n + 1$ or $2n$ to represent the odd numbers and even numbers, then

$$Num_{Valid} = \begin{cases} \dfrac{(2n+1)^2 + 3(2n+1) - 2}{2} = 2n^2 + 5n + 1, \text{where } n \geq 0 \\ \dfrac{(2n)^2 + 3(2n) - 2}{2} = 2n^2 + 3n - 1, \text{where } n \geq 1 \end{cases}$$

Therefore, when n is a natural number, Num_{Valid} is always a natural number. In this case, the number of the matrix index is 23, so the number of valid combinations can be obtained as 298, which means only 298 out of all the combinations are valid under the testing scenario. As a result, the reachset coverage in key states should be calculated by comparing it with the 298 valid combinations. In Fig 67, the reachset coverage in key states is 55.7%, which is an

acceptable number for 5000 seconds of testing.

However, the purpose of testing is to cover as many possibilities as possible, to reduce the chance of missing errors caused by uncovered possibilities. To achieve that goal, the author implemented a series of experiments to improve the coverage and to find out the elements which may influence it. Based on the given testing environment, two elements were discovered to have an impact on the reachset coverage in key states, testing time and the intensity of the train interaction. Since the testing platform is designed to cause diversity, to cover as many possibilities as are contained in the specification model, it has more chance of covering more possibilities when it is given more time. As a result, testing time becomes the most influential factor of the reachset coverage in key states, which is shown by Fig 69 and Fig 70:

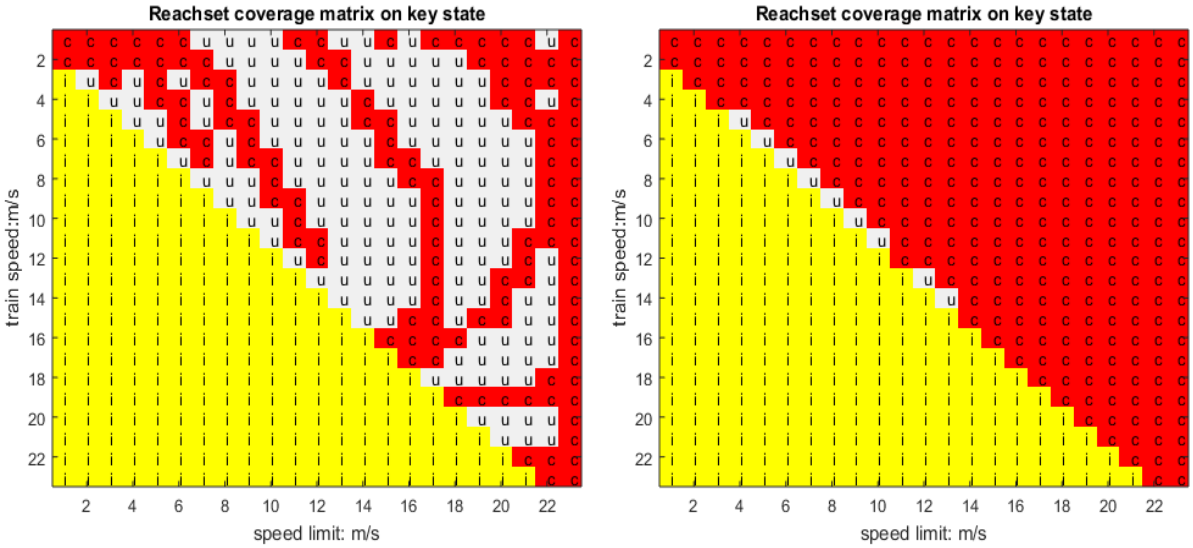


Fig 69 Coverage matrices for different testing times (1000 seconds on the left and 50000 seconds on the right)

As shown in Fig 69, the reachset coverage of 1000-second testing is 42.0%, and the reachset coverage of 50000-second testing is 97.0%. The influence of testing time is obvious, and a longer testing time tends to achieve higher reachset coverage in key states. The relation of testing time and corresponding reachset coverage is shown by Fig 70:

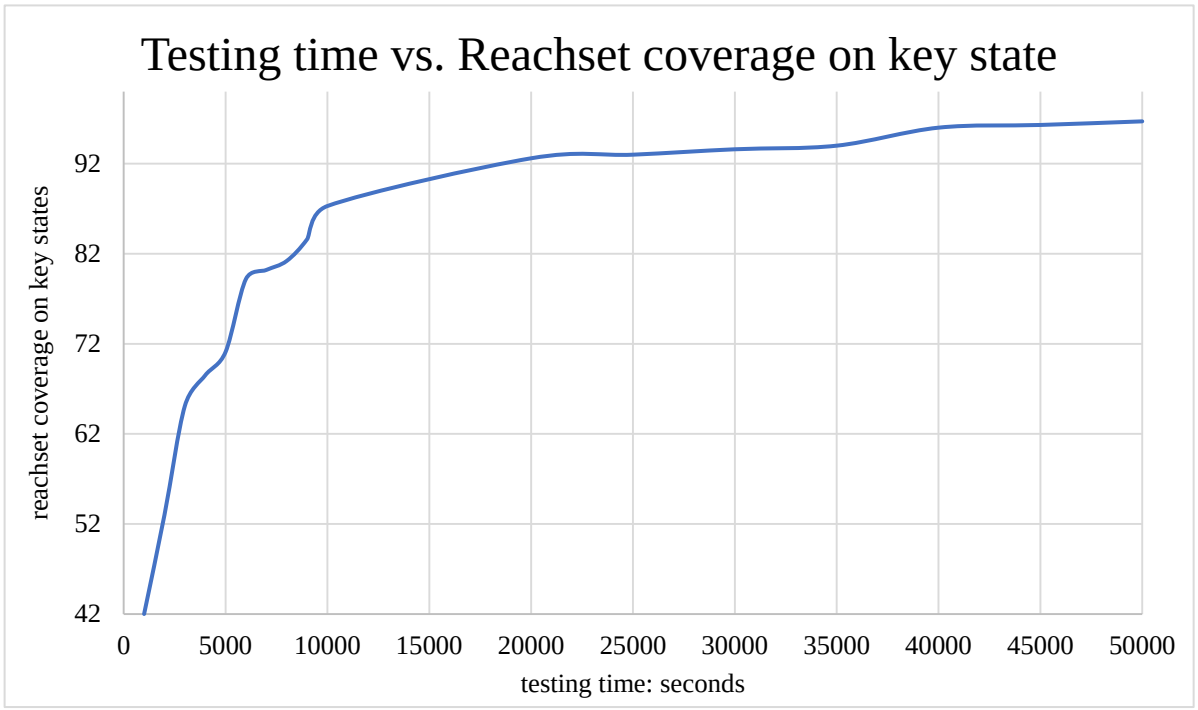


Fig 70 Relation between reachset coverage and testing time

The graph in Fig 70 indicates the tendency of reachset coverage in key states to vary with increasing testing time. Since randomness exists in every individual testing process, Fig 70 can simply prove that a longer testing time tends to obtain better reachset coverage. When the testing time is longer than 25000 seconds, the reachset coverage tends to reach the limitation which is approximately 97%. Furthermore, the growth rate of the reachset coverage slows down significantly after the coverage is above 92% and the testing time is longer than 20000 seconds. Therefore, simply extending the testing time is not the most efficient way to reach

the maximum reachset overage in key states, and other influential factors should be reconfigured to improve it.

Another factor influential in reachset coverage is the intensity of interaction of the two trains. With relatively weak interaction between the front train and the SUT train, the SUT train's MA is less influenced by the front train, which makes it achieve fewer combinations of train speed and speed limit, as shown by Fig 71:

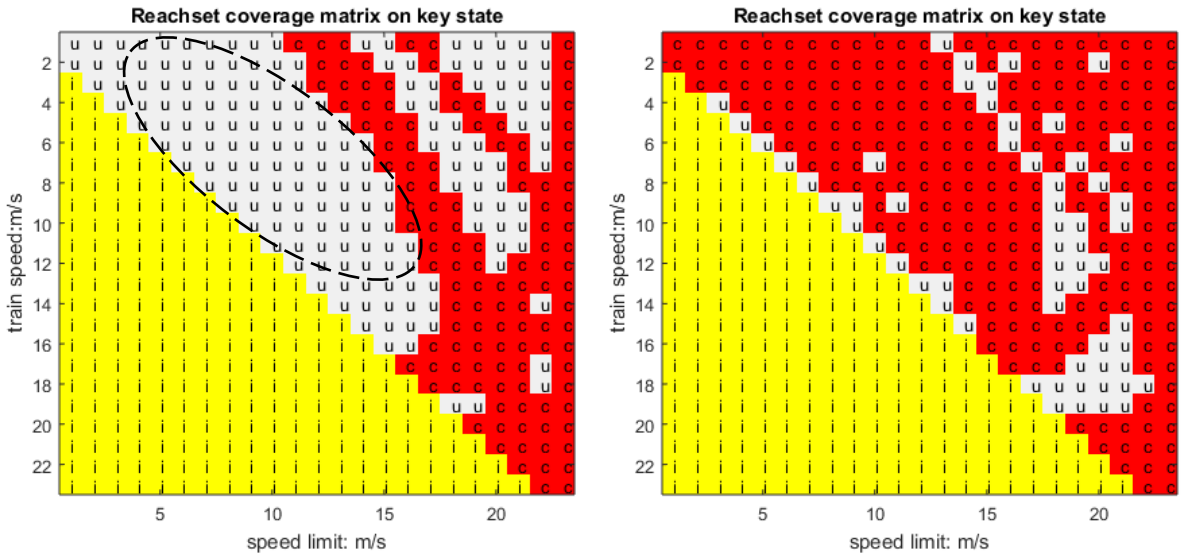


Fig 71 Reachset coverage under different train interaction intensities (weak interaction on the left and strong interaction on the right)

As indicated by Fig 71, weak interaction leads to a poor reachset coverage because the marked area is missed. The reason is that when the front train is far away from the SUT train, the SUT train has fewer opportunities to exceed the speed limit influenced by the front train position, which means that most of the speed limit of the SUT train is determined by the line speed limit which is a constant value. With the same testing time of 20000 seconds, the

left-hand graph has the low speed limit area missing which means the front train is relatively further away from the SUT train than in the right-hand graph. Therefore, the tester should configure a stronger interaction between the front train and the SUT in the testing implementation to achieve a higher reachset coverage in key states. In this thesis, the author strengthened the interactions by increasing the service delay of the front train, which means that the SUT train has more opportunities to approach the front train and to be blocked by it. In the right-hand graph of Fig 71, a service delay of 30 seconds is inserted for the front train, which improves the reachset coverage significantly for the same testing time.

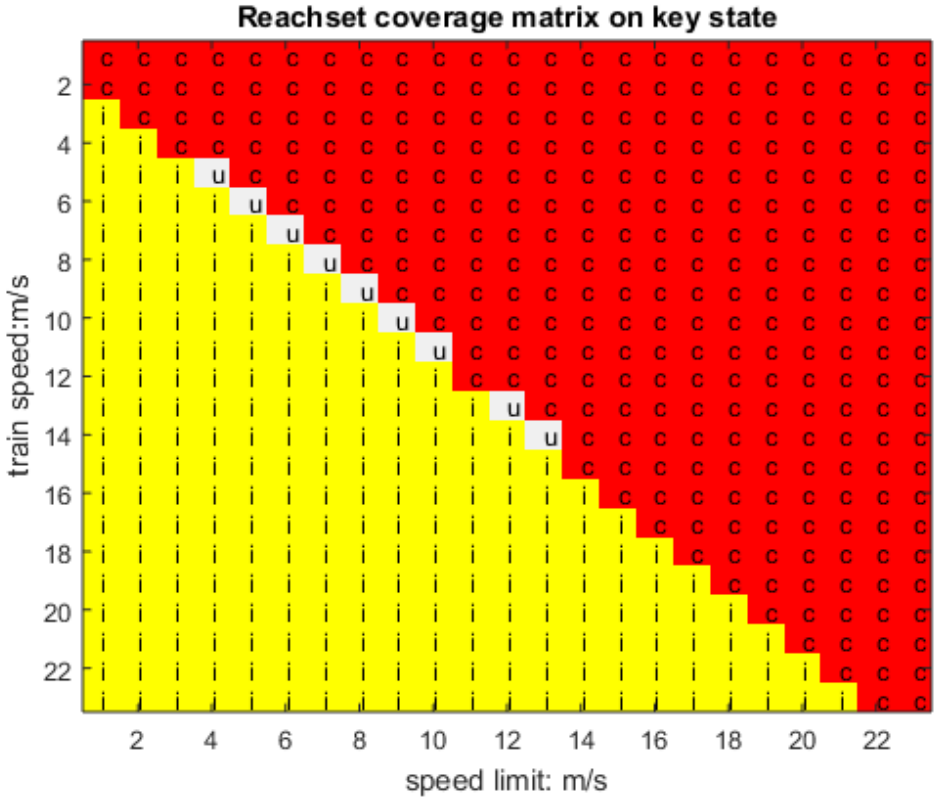


Fig 72 Reachset coverage matrix for the maximum percentage of 97%

From Fig 70, which describes the relationship between testing and reachset coverage in key

states, improvements in the reachset coverage are no longer obvious when the testing time is longer than 25000 seconds. The maximum coverage of 97% tends to be achieved when the testing time reaches 50000 seconds, which is a relatively long time. Spending more time on testing to cover the missing 3% of combinations is not cost-effective. Therefore, the author covered the missing combinations by reconfiguring the testing scenario.

From Fig 72, it is obvious that all the missing combinations are located on the boundary between valid and invalid combinations, meaning that the train speed is $1\,\mathrm{m/s}$ faster than the speed limit. The reason that these combinations are missed is that the SUT train does not overspeed seriously under certain values of the speed limit. Therefore, to cover the missing combinations, the author directly set the line speed limit to the certain values for which the combinations were missed, using the constant line speed limit to present the dynamic speed limit determined from the MA. With the purpose of verifying that all the possibilities contained in the specification are covered, this straightforward method is acceptable to prove that no corresponding errors of missed combinations are missed by the testing platform in the testing process. As a result, the author manually set the line speed limit to the speed limit values of the missed combinations and checked whether the SUT could make the correct decision in the configured situations. Fig 73 and Fig 74 show the verification results for one of the nine missed combinations, '**SPEED**=5, **speedlim**=4':

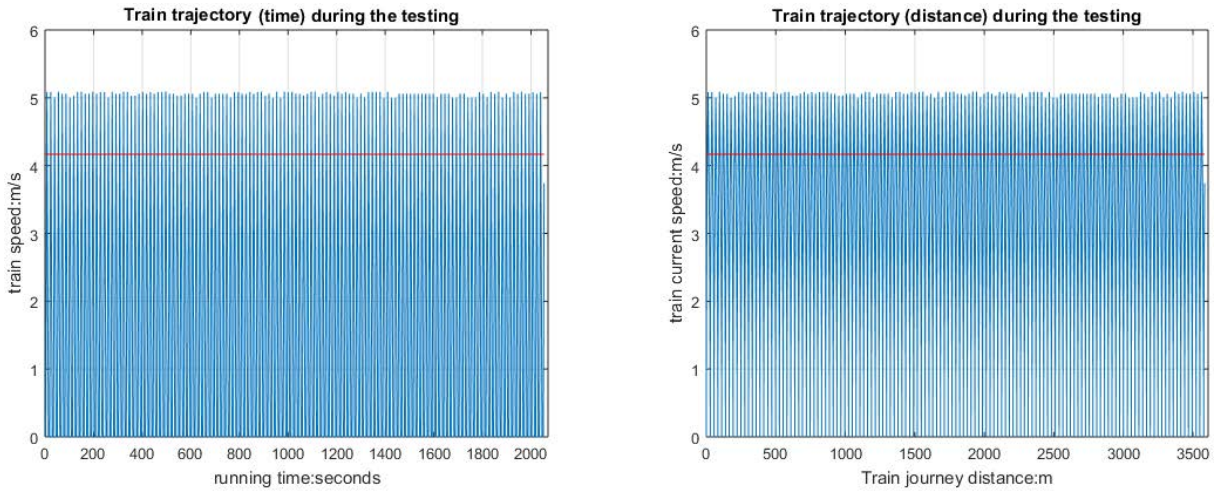


Fig 73 Train trajectory for verifying the missed combination '**SPEED**=5, **speedlim**=4'

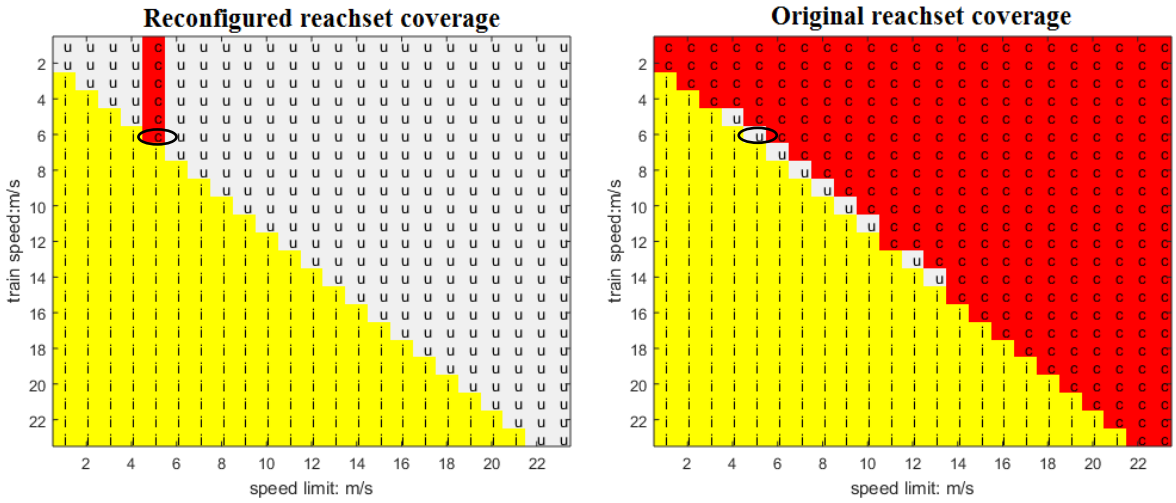


Fig 74 Verification result for the missed combination '**SPEED**=5, **speedlim**=4'

As shown by Fig 72 and Fig 73, the missed combination '**SPEED**=5, **speedlim**=4' is covered in the reconfigured testing scenario where the line speed limit is set to a constant $4\ \mathrm{m/s}$. As indicated by the left-hand graph of Fig 74, the originally missed combination '6,5' (corresponding to '**SPEED**=5, **speedlim**=4' in the real testing results) is covered in the reconfigured testing scenario without detecting any inconsistencies between the SUT and the

specification, which can be proven by Fig 73. By applying the same verification method to the other missed combinations, all the combinations originally missed are covered eventually, as summarised in Table 10:

Missed combination	Verification conclusion	Verification time
SPEED=4, speedlim=3	Verified	2000 seconds
SPEED=5, speedlim=4	Verified	2000 seconds
SPEED=6, speedlim=5	Verified	2000 seconds
SPEED=7, speedlim=6	Verified	2000 seconds
SPEED=8, speedlim=7	Verified	2000 seconds
SPEED=9, speedlim=8	Verified	2000 seconds
SPEED=10, speedlim=9	Verified	2000 seconds
SPEED=12, speedlim=11	Verified	2000 seconds
SPEED=13, speedlim=12	Verified	2000 seconds

Table 10 Summary of the verification of missed combinations

The verification results indicate that the SUT complies with the specification for those missed combinations. However, the limitation of the verification method is obvious in that it can only verify a limited number of missed combinations, and it becomes time-consuming when too many combinations are missed. Therefore, it is essential for the testing platform to cover as high a percentage as possible in one testing process. In this case, the maximum reachset coverage in key states is 97%, leaving nine combinations which need 18000 seconds to be verified manually, which is efficient compared with extending the testing time.

However, simply counting the covered and missed combinations cannot fully illustrate the coverage ability of the testing platform because covering a combination 1000 times is no different to covering it only once. To indicate the covering tendency of the testing platform for every individual combination, the author extended the reachset coverage in key states to not

only check if a combination is covered in the testing results but also count the number of times that a combination is covered. Therefore, the covering tendency of the testing platform in a certain testing scenario can be obtained, as shown by Fig 75:

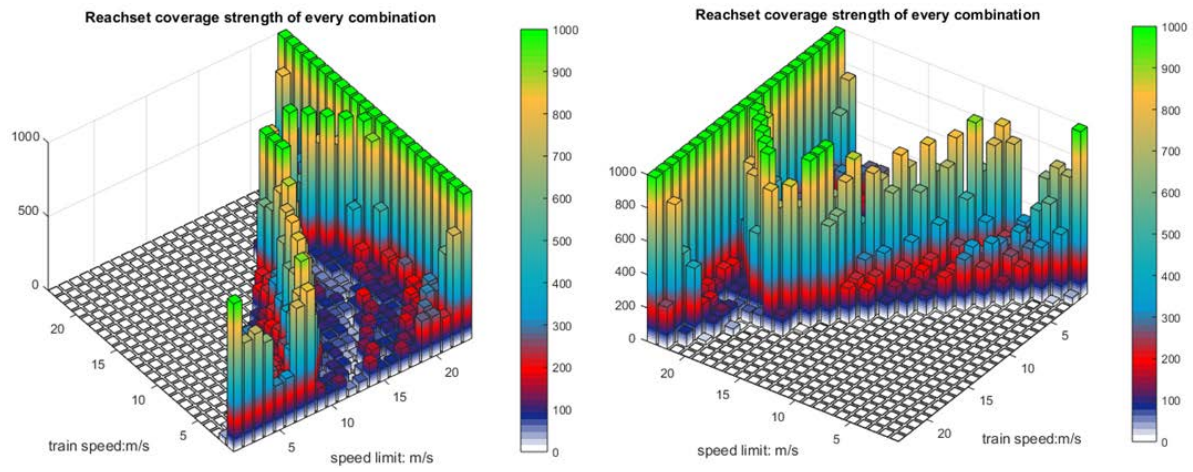


Fig 75 Reachset coverage strength in key states for every combination

As shown by the two 3D bar graphs in Fig 75, the reachset coverage strength is obviously high in two areas; one is on the line where the speed limit equals the line speed limit of $22\ \mathrm{m/s}$, and the other is near the matrix diagonal where the train speed is 4 to $5\ \mathrm{m/s}$ below the speed limit, which can be clearly illustrated by the yellow line in Fig 76:

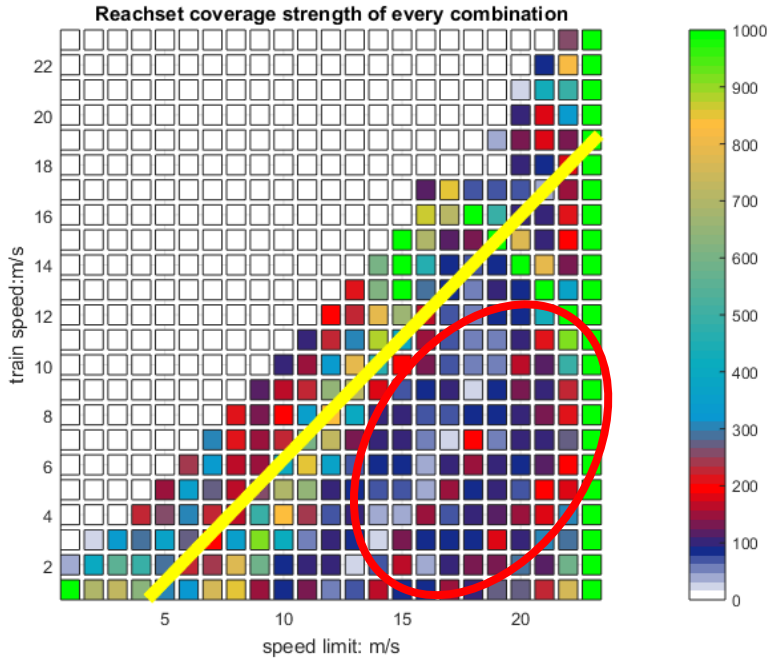


Fig 76 Planar figure of the 3D bar graph of coverage strength

As indicated by Fig 75 and Fig 76, the maximum number of times the combination is covered appears in the light green area in Fig 76. On the contrary, the minimum number of times a combination is covered is in the red ellipse where most of the combinations are covered less than 100 times in the whole testing process. As seen from Fig 76, the maximum number of cover times can be achieved when the speed limit equals the line speed limit. The reason is that the SUT train mostly overspeeds when it exceeds the line speed limit under the current configuration of the testing scenario. This phenomenon is reasonable in the current testing scenario where the interactions between the front train and the SUT train are not strong enough to cover the area marked with the red ellipse. To cover the marked area, the SUT train movements must be influenced more strongly by the front train, which can be achieved by decreasing the top speed of the front train, because a slower front train makes the SUT train's

speed limit lower than the line speed limit. Improved reachset coverage strength is shown in Fig 77:

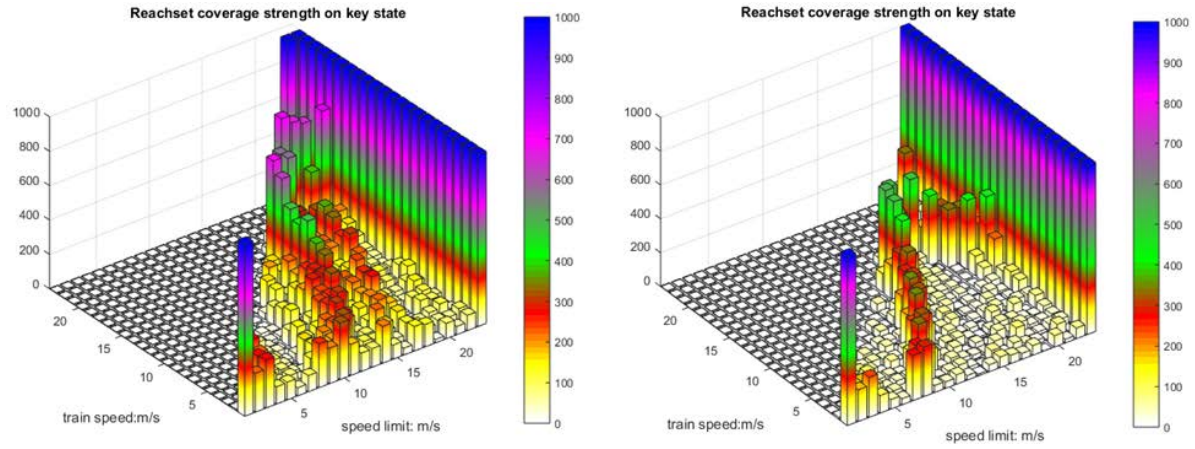


Fig 77 Improved coverage strength with a lower top speed of the front train

In the left-hand graph in Fig 77, the top speed of the front train is set to be $50\ \mathrm{km/h}$ which is lower than that in the original configuration (80 km/h). As a result, the impact of the front train on the MA of the SUT train becomes much stronger. Therefore, the reachset coverage strength significantly improves when the speed limit is $[10,\ 20]\ \mathrm{m/s}$. Furthermore, within the same testing time of 20000 seconds, the left-hand graph achieves a reachset coverage of 95.6% while that in the right-hand one is 90.9%, which proves that the top speed of the front train can influence performance of the reachset coverage in key states. A well-configured testing scenario can make the testing process cover all the combinations more evenly under the same testing time, which means that the testing efficiency is improved.

6.4 Summary

In this chapter, the effectiveness and performance of the simulation combined MBT platform

were verified by the author. According to the verification results, the testing platform is superior to the traditional manual testing methods and the traditional offline testing introduced in Chapter 2. Compared with traditional online MBT methods, the simulation combined MBT platform can automatically test more complex SUTs in an HIL environment, which allows off-site testing. With a verified specification model, the ability of the testing platform to detect known errors was verified by implementing a series of mutation tests. The results of the mutation testing indicate that the testing platform can find most of the common errors which can be found in TCS system testing. For unknown errors, the reachset conformance relation proves that the testing platform does not violate the safety properties required by the specifications. The verification results indicate that the testing platform does not miss known or unknown errors which can lead the system into dangerous situations, such as overspeed and collision.

The effectiveness verification proves that the testing platform does not miss errors in the SUT, and the performance verification proves that the testing platform can cover all the possibilities contained in the specification model. The verification results show that the testing platform could cover 100% of traces and variables in the abstract model with sufficient search depth, performing better than traditional manual testing which covers a single trace, and offline testing which covers part of the traces and variables because of the high degree of complexity of the abstract model. Furthermore, to determine whether the SUT VOBC can make the correct decision under any accessible circumstance in the specification, the author introduced reachset coverage in key states to verify whether the testing platform can cover all possible

combinations of train speed and speed limit. The verification results show that the testing platform can cover a maximum of 97% of the possible combinations, and only 3% is lost in one-time testing. By reconfiguring the testing scenario, the missed combinations can be covered in another period of testing, to reach 100% coverage. In summary, with the validated specification model and verified effectiveness and performance, simulation combined MBT is proven to be effective for detecting errors with better performance.

7 Conclusion

7.1 Conclusion

In this thesis, the author has proposed a simulation combined MBT methodology and the implementation, which can perform automatic off-site testing of TCSs.

Firstly, MBT methods were introduced as the solution for automatic TCS testing where state explosion and processing power limit the testing by conventional means. To address the limitations of current MBT methods, the simulation combined MBT method was proposed to overcome the difficulties of testing TCSs using existing MBT methods. The proposed methodology has the potential to be applied to test different types of TCSs because of the shared common functional features.

To achieve automatic functional testing of TCSs, the modelling theory of simulation combined MBT, named SCTIOTS, was explained in detail. Through formula derivation, SCTIOTS was theoretically proven to be capable of describing system behaviour in a two-model-combined structure, which provides the possibility of realising simulation combined MBT. Afterwards, implementation of simulation combined MBT was introduced by developing a simulation combined MBT platform, which is an integrated testing platform for automating TCS functional testing in an HIL environment. Essential components of the testing platform were introduced, including the modelling tools, test tools, I/O sequence manager, HIL environment and data interfaces.

To prove the feasibility of the developed simulation combined MBT platform, two case studies were undertaken. A VOBC of the CBTC system was chosen to be the SUT, and its overspeed protection and train location functions were tested in the two cases. The single train scenario concentrated on explaining the built components of the simulation combined MBT platform, including the internal function of each component. The multiple train scenario was designed to reveal whether the VOBC can protect the train operating safety when travelling on the same line as other vehicles. The testing results for both cases were recorded by the testing platform through the whole testing procedure; they indicate that the proposed simulation combined MBT methodology and the developed platform are effective to undertake functional testing for TCSs.

Lastly, the developed testing platform was validated and verified to prove its effectiveness and performance. Firstly, the TA model was verified by an integrated verification tool in UPPAAL. The safety and liveness properties were validated to see whether there is any error which can lead to wrong testing results. All the safety and liveness properties passed the validation. Afterwards, the testing platform was verified to inspect whether it can find known errors via six mutation tests. Furthermore, to inspect whether the testing platform can miss any unknown errors which could lead the SUT into dangerous situations, the testing results were verified to inspect whether the SUT complies with the reachset conformance relation. The verification results indicate that the testing platform can detect known errors and does not miss unknown errors. The last verification was to verify whether the testing platform achieves better results than existing testing methods. The contrast object chosen was an offline test

generation tool integrated in UPPAAL, which is capable of generating test cases according to test selection criteria. The comparison results indicate that the testing platform has better coverage than the offline test generation tool, as the simulation combined MBT platform can achieve 100% coverage on variables and traces within feasible search depth while the offline test generation can only achieve 91% trace coverage and 61% variable coverage. Lastly, the author explored whether the performance of the testing platform could potentially be improved. The concept of reachset coverage in key states was introduced to express the ability of the testing platform to cover all possibilities. The maximum reachset coverage in key states which can be achieved is 97%; 3% is lost due to inappropriate configuration of the test scenario. By adjusting the test scenario to strengthen the interaction between the three trains, the full reachset could be covered, which indicates that the coverage performance can be improved by well-configured test scenarios.

From the testing results derived from the cases in Chapter 5, and the validation and verification results obtained in Chapter 6, the proposed simulation combined MBT method and the developed simulation combined MBT platform are proven to be feasible and effective for functional testing of TCSs. The testing platform can detect errors contained in the SUT with a better coverage performance than existing methods.

7.2 Contribution

The contribution of the author's research can be summarised as follows:

- The author has combined formal methods and simulation technologies in an HIL testing

framework and proposed a simulation combined MBT methodology to improve the current functional testing methods for TCSs.

- Based on the existing MBT modelling theory, the author has developed a modelling approach named SCTIOTS, which supports formal modelling combined with simulation.
- Based on the proposed modelling method, a simulation combined MBT platform has been developed for methodology implementation. The testing and verification results indicate that the testing platform is effective.
- The reachset conformance relation has been introduced to verify the coverage performance of the testing platform. The reachset conformance relation in key states quantifies the coverage of online testing results by discretising the valid variable combinations in key states. Furthermore, it shows that the test scenario configurations have impact on test efficiency performance and coverage performance.

7.3 Future Work

The testing results for the case studies, and verification results in Chapter 6 indicate the benefits of applying simulation combined MBT to test TCSs. Likewise, it reveals the potential to improve the proposed research by extending it in the following directions:

- Explore the possibilities of adopting a hierarchical structure in formal modelling to improve the modelling efficiency for large complex systems without losing system information.
- The operating principle of online testing leads to a fatal flaw in testing performance. In

online testing, inputs are randomly chosen without guidance from the test selection criteria because the possibility space is too large to be restored in the computer memory. Based on the introduced reachset conformance relation, the possibility space can be reduced. The author aims to improve the online test algorithm by adding an input selection function, to achieve optimised coverage performance within a shorter testing time.

- Currently, MBT methods still rely on humans to build formal models according to specification requirements in natural languages. The author aims to develop a modelling tool which supports the building of formal models by analysing formatted specification requirements in natural languages. As a result, the errors caused by human factors can be isolated so that testing efficiency and accuracy can be further improved.
- The current online test generation algorithm adopted by the author in this thesis is a 32-bit program that utilises no more than 4 GB of memory, which limits the algorithm capability of analysing large complex models. An improved online test algorithm capable of handling large complex models could be further developed.
- The results in the thesis shows that test scenarios have an unneglectable influence on coverage performance and test efficiency. Developing a testing scenario optimiser which interacts with the HIL test environment and the SUT along with the online MBT algorithm could be further studied to improve the efficiency of the proposal simulation combined MBT methodology.

Appendix: Publications

The articles published during the author's PhD study are presented below:

[1] W. Yuemiao, C. Lei, W. Jinwen, D. Kirkwood, X. Qian, J. Lv, *et al.*, "On-line conformance testing of the Communication-Based Train Control (CBTC) system," in *2016 IEEE International Conference on Intelligent Rail Transportation (ICIRT)*. Piscataway, NJ: IEEE, 2016, pp. 328–333.

[2] W. Jinwen, X. Qian, D. Kirkwood, W. Yuemiao, C. Lei, L. Jidong, *et al.*, "Verification of metro track signalling layout based on microscopic simulation," in *2016 IEEE International Conference on Intelligent Rail Transportation (ICIRT)*. Piscataway, NJ: IEEE, 2016, pp. 494–499.

[3] Yuemiao Wang, Lei Chen*, David Kirkwood, Jidong Lv, Clive Roberts, *et al.*, "Hybrid Online Model-Based Testing for Communication-Based Train Control Systems," in *IEEE Intelligent Transportation System Magazine*. Minor Correction Decision Received.

References

[1] N. Zhao, L. Chen, Z. Tian, C. Roberts, S. Hillmansen, and J. Lv, "Field test of train trajectory optimisation on a metro line," IET Intelligent Transport Systems, vol. 11, pp. 273-281, 2017.

[2] J. Wang, J. Wang, C. Roberts, and L. Chen, "Parallel Monitoring for the Next Generation of Train Control Systems," IEEE Transactions on Intelligent Transportation Systems, vol. 16, pp. 330-338, 2015.

[3] H. Wang, F. Schmid, L. Chen, C. Roberts, and T. Xu, "A Topology-Based Model for Railway Train Control Systems," IEEE Transactions on Intelligent Transportation Systems, vol. 14, pp. 819-827, 2013.

[4] A. E. Haxthausen and J. Peleska, "Formal development and verification of a distributed railway control system," IEEE Transactions on Software Engineering, vol. 26, pp. 687-701, 2000.

[5] J. Wang, "Chapter 3 - Theory System and Framework of High-Speed Railway Train Operation Safety," in Safety Theory and Control Technology of High-Speed Train Operation, London: Academic Press, 2018, pp. 79-123.

[6] UNISIG, System Requirements Specifcation (SUBSET-026). ERTMS, 2016.

[7] UNISIG, FFFIS STM test cases of Functional identity (SUBSET-074-2). ERTMS, 2015.

[8] L. Zhu, F. R. Yu, B. Ning, and T. Tang, "Design and Performance Enhancements in

Communication-Based Train Control Systems With Coordinated Multipoint Transmission and Reception," IEEE Transactions on Intelligent Transportation Systems, vol. 15, pp. 1258-1272, 2014.

[9] L. Zhu, F. R. Yu, B. Ning, and T. Tang, "Communication-Based Train Control (CBTC) Systems With Cooperative Relaying: Design and Performance Analysis," IEEE Transactions on Vehicular Technology, vol. 63, pp. 2162-2172, 2014.

[10] J. Młyńczak, A. Toruń, and L. Bester, "European Rail Traffic Management System (ERTMS)," in Intelligent Transportation Systems – Problems and Perspectives, A. Sładkowski and W. Pamuła, Eds. Cham: Springer International Publishing, 2016, pp. 217-242.

[11] M. Ghazel, "A Control Scheme for Automatic Level Crossings Under the ERTMS/ETCS Level 2/3 Operation," IEEE Transactions on Intelligent Transportation Systems, vol. 18, pp. 2667-2680, 2017.

[12] ETSI, "Railways Telecommunications (RT);Global System for Mobile communications (GSM); Detailed requirements for GSM operation on Railways". Sophia Antipolis: France, 2016.

[13] H. Dong, B. Ning, B. Cai, and Z. Hou, "Automatic Train Control System Development and Simulation for High-Speed Railways," IEEE Circuits and Systems Magazine, vol. 10, pp. 6-18, 2010.

[14] W. Yuemiao, C. Lei, W. Jinwen, D. Kirkwood, X. Qian, J. Lv, et al., "On-line

conformance testing of the Communication-Based Train Control (CBTC) system," in 2016 IEEE International Conference on Intelligent Rail Transportation (ICIRT). Piscataway, NJ: IEEE, 2016, pp. 328-333.

[15] M. Idirin, X. Aizpurua, A. Villaro, J. Legarda, and J. Melendez, "Implementation Details and Safety Analysis of a Microcontroller-based SIL-4 Software Voter," IEEE Transactions on Industrial Electronics, vol. 58, pp. 822-829, 2011.

[16] J. Wang, "Chapter 4 - System-Level “Fail-Safe”," in Safety Theory and Control Technology of High-Speed Train Operation. London: Academic Press, 2018, pp. 125-144.

[17] P. Bourque, R. E. Fairley, and IEEE Computer Society, Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0. Los Alamitos, CA: IEEE Computer Society Press, 2014.

[18] P. Samuel, R. Mall, and A. K. Bothra, "Automatic test case generation using unified modeling language (UML) state diagrams," IET Software, vol. 2, pp. 79-93, 2008.

[19] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "MobiGUITAR: Automated Model-Based Testing of Mobile Apps," IEEE Software, vol. 32, pp. 53-59, 2015.

[20] A. Belinfante, J. Feenstra, R. G. de Vries, J. Tretmans, N. Goga, L. Feijs, et al., "Formal Test Automation: A Simple Experiment," in Testing of Communicating Systems: Methods and Applications, G. Csopaki, S. Dibuz, and K. Tarnay, Eds. Boston,

MA: Springer US, 1999, pp. 179-196.

[21] E. Dincel, O. Eris, and S. Kurtulan, "Automata-Based Railway Signaling and Interlocking System Design," IEEE Antennas and Propagation Magazine. 308 - 319.

[22] T. P. Parker and G. L. Harrison, "Quality improvement using environmental stress testing," AT&T Technical Journal, vol. 71, pp. 10-23, 1992.

[23] C. Zoeller, M. A. Vogelsberger, R. Fasching, W. Grubelnik, and T. M. Wolbank, "Evaluation and Current-Response-Based Identification of Insulation Degradation for High Utilized Electrical Machines in Railway Application," IEEE Transactions on Industry Applications, vol. 53, pp. 2679-2689, 2017.

[24] D. Chisnell, "Usability testing: Taking the experience into account," IEEE Instrumentation & Measurement Magazine, vol. 13, pp. 13-15, 2010.

[25] I. Molyneaux, The Art of Application Performance Testing: Help for Programmers and Quality Assurance. Sebastopol: O'Reilly Media, Inc., 2009.

[26] I. Schieferdecker, "Model-Based Testing," IEEE Software, vol. 29, pp. 14-18, 2012.

[27] J. Zander, I. Schieferdecker, and P. J. Mosterman, Model-based testing for embedded systems. Boca Raton, FL: CRC press, 2011.

[28] A. Ferrari, A. Fantechi, S. Gnesi, and G. Magnani, "Model-Based Development and Formal Methods in the Railway Industry," IEEE Software, vol. 30, pp. 28-34, 2013.

[29] L. Padgham, Z. Zhang, J. Thangarajah, and T. Miller, "Model-Based Test Oracle Generation for Automated Unit Testing of Agent Systems," IEEE Transactions on

Software Engineering, vol. 39, pp. 1230-1244, 2013.

[30] M. Utting and B. Legeard, Practical model-based testing: a tools approach. San Francisco: Morgan Kaufmann, 2007.

[31] S. Elbaum, H. N. Chin, M. B. Dwyer, and M. Jorde, "Carving and Replaying Differential Unit Test Cases from System Test Cases," IEEE Transactions on Software Engineering, vol. 35, pp. 29-45, 2009.

[32] N. Nisan and S. Schocken, "Test Scripting Language," in The Elements of Computing Systems:Building a Modern Computer from First Principles. Cambridge, MA, London: MIT Press, 2008, pp. 297-313.

[33] V. Garousi and M. Felderer, "Developing, Verifying, and Maintaining High-Quality Automated Test Scripts," IEEE Software, vol. 33, pp. 68-75, 2016.

[34] ISO/IEC/IEEE, ISO/IEC/IEEE 29119-5 First Edition 2016-11-15: ISO/IEC/IEEE International Standard - Software and Systems Engineering -- Software testing -- Part 5: Keyword-Driven Testing. IEEE, 2016, pp. 1–69.

[35] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The Oracle Problem in Software Testing: A Survey," IEEE Transactions on Software Engineering, vol. 41, pp. 507-525, 2015.

[36] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded Model Checking Using Satisfiability Solving," Formal Methods in System Design, vol. 19, pp. 7-34, 2001.

[37] M. A. Noureddine and F. A. Zaraket, "Model Checking Software with First Order

Logic Specifications Using AIG Solvers," IEEE Transactions on Software Engineering, vol. 42, pp. 741-763, 2016.

[38] J. Z. Gao, J. Tsao, Y. Wu, Testing and Quality Assurance for Component-Based Software. Boston, MA: Artech House, Inc., 2003.

[39] IEEE, IEEE Standard for Software Unit Testing: ANSI/IEEE Std 1008-1987. New York: IEEE, 1987, pp. 1-28.

[40] Z. J. Li, H. F. Tan, H. H. Liu, J. Zhu, and N. M. Mitsumori, "Business-process-driven gray-box SOA testing," IBM Systems Journal, vol. 47, pp. 457-472, 2008.

[41] K. Li, X. Yao, and D. Chen, "HAZOP Study on the CTCS-3 Onboard System," IEEE Transactions on Intelligent Transportation Systems, vol. 16, pp. 162-171, 2015.

[42] A. En-Nouaary, R. Dssouli, and F. Khendek, "Timed Wp-method: testing real-time systems," IEEE Transactions on Software Engineering, vol. 28, pp. 1023-1038, 2002.

[43] S. C. Paiva and A. Simao, "Generation of complete test suites from mealy input/output transition systems," Formal Aspects of Computing, vol. 28, pp. 65-78, March 01 2016.

[44] S. Schneider, The B-Method: An Introduction. Basingstoke: Palgrave, 2001.

[45] D. Cansell and D. Mery, "Tutorial on the event-based B method: Concepts and Case Studies," presented at the 26th IFIP WG 6.1 International Conference on Formal Methods for Network and Distributed Systems, Paris, France, 2006.

[46] N. A. Zafar, "Formal specification and validation of railway network components using Z notation," IET Software, vol. 3, pp. 312-320, 2009.

[47] A. Giorgetti, J. Groslambert, J. Julliand, and O. Kouchnarenko, "Verification of class liveness properties with java modelling language," IET Software, vol. 2, pp. 500-514, 2008.

[48] M. Barnett, K. R. M. Lenio, and W. Schulte, "The Spec# Programming System: an overview," in Construction and Analysis of Safe, Secure, and Interoperable Smart Devices. CASSIS 2004, G. Barthe, L. Burdy, M. Huisman, J. L. Lanet, and T. Muntean, Eds. (Lecture Notes in Computer Science, vol. 3362). Berlin, Heidelberg: Springer, 2004.

[49] G. Babin, Y. Aït-Ameur, and M. Pantel, "Web Service Compensation at Runtime: Formal Modeling and Verification Using the Event-B Refinement and Proof Based Formal Method," IEEE Transactions on Services Computing, vol. 10, pp. 107-120, 2017.

[50] R. M. Hierons, "Testing from Partial Finite State Machines without Harmonised Traces," IEEE Transactions on Software Engineering, vol. 43, pp. 1033-1043, 2017.

[51] R. Alur and D. L. Dill, "A theory of timed automata," Theor. Comput. Sci., vol. 126, pp. 183-235, 1994.

[52] H. B. Mokadem, B. Berard, V. Gourcuff, O. D. Smet, and J. M. Roussel, "Verification of a Timed Multitask System With Uppaal," IEEE Transactions on Automation Science and Engineering, vol. 7, pp. 921-932, 2010.

[53] L. Yang, J. Daming, D. Shenghua, and L. Zhengjiao, "Hierarchical modeling and

analysis of TCC subsystem in CTCS level 3 using UPPAAL," in 2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC), 2016, pp. 713-718.

[54] M. Li and R. Kumar, "Automated test generation and error localisation for Simulink/Stateflow modelled systems using extended automata," IET Cyber-Physical Systems: Theory & Applications, vol. 1, pp. 95-107, 2016.

[55] M. R. Blackburn and R. D. Busser, "T-VEC: a tool for developing critical systems," in Proceedings of the Eleventh Annual Conference on Computer Assurance, 1996. COMPASS '96, Systems Integrity. Software Safety. Process Security. New York: IEEE, 1996, pp. 237–249.

[56] M. Blackburn, R. Busser, A. Nauman, R. Knickerbocker, and R. Kasuda, "Mars Polar Lander fault identification using model-based testing," in Proceedings of the Eighth IEEE International Conference on Engineering of Complex Computer Systems, 2002. Los Alamitos, CA: IEEE Computer Society, 2002, pp. 163–169.

[57] M. R. Blackburn, "Using models for test generation and analysis," in Proceedings of the 17th DASC/AIAA/IEEE/SAE Digital Avionics Systems Conference, vol. 1. Piscataway, NJ: IEEE, 1998, pp. C45/1–C45/8.

[58] M. Blackburn, R. D. Busser, and J. S. Fontaine, "Automatic generation of test vectors for SCR-style specifications," in Proceedings of the 12th Annual Conference on Computer Assurance, 1997. COMPASS '97. Are We Making Progress Towards

Computer Assurance? New York: IEEE, 1997, pp. 54–67.

[59] E. Rudolph, P. Graubmann, and J. Grabowski, "Tutorial on message sequence charts," Computer Networks and ISDN Systems, vol. 28, pp. 1629–1641, 1996.

[60] C. A. R. Hoare, Communicating Sequential Processes. New York: Prentice Hall International 2015.

[61] Z. Ding, M. Jiang, and M. Zhou, "Generating Petri Net-Based Behavioral Models From Textual Use Cases and Application in Railway Networks," IEEE Transactions on Intelligent Transportation Systems, vol. 17, pp. 3330-3343, 2016.

[62] L. Jidong, W. Haifeng, L. Hongjie, Z. Lu, and T. Tao, "A model-based test case generation method for function testing of train control systems," in 2016 IEEE 19th International Conference on Intelligent Rail Transportation (ICIRT). Piscataway, NJ: IEEE, 2016, pp. 334–346.

[63] J. Magott, "Performance evaluation of communicating sequential processes (CSP) using Petri nets," IEE Proceedings E - Computers and Digital Techniques, vol. 139, pp. 237-241, 1992.

[64] B. Nevio and M. Zorzi, "Markov chains theory," in Principles of Communications Networks and Systems. Chichester, West Sussex: Wiley Telecom, 2011, p. 816.

[65] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson, "Model-based testing of object-oriented reactive systems with Spec Explorer," Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected

Papers, R. M. Hierons, J. P. Bowen, and M. Harman, Eds. Berlin, Heidelberg: Springer Verlag, 2008.

[66] I. B. Bourdonov, A. S. Kossatchev, V. V. Kuliamin, and A. K. Petrenko, "UniTesK test suite architecture," in Proceedings of FME 2002: Formal Methods—Getting IT Right: International Symposium of Formal Methods Europe Copenhagen, Denmark, July 22–24, 2002, L.-H. Eriksson and P. A. Lindsay, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 77–88.

[67] V. V. Kuliamin, A. K. Petrenko, A. S. Kossatchev, and I. B. Burdonov, "The UniTesK Approach to Designing Test Suites," Programming and Computer Software, vol. 29, pp. 310-322, 2003.

[68] J. Warmer and A. Kleppe, The Object Constraint Language: Getting Your Models Ready for MDA. Boston, MA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[69] H. M. Tahir, M. Nadeem, and N. A. Zafar, "Specifying electronic health system with Vienna development method specification language," in 2015 National Software Engineering Conference (NSEC). Piscataway, NJ: IEEE, 2015, pp. 61–66.

[70] J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef, Validated Designs For Object-oriented Systems. London: Springer-Verlag TELOS, 2005.

[71] J. s. Lee and P. l. Hsu, "Statechart-based representation of hybrid controllers for vehicle automation," IEE Proceedings - Intelligent Transport Systems, vol. 153, pp. 253-258, 2006.

[72] S. Arifiani and S. Rochimah, "Generating test data using ant Colony Optimization (ACO) algorithm and UML state machine diagram in gray box testing approach," in 2016 International Seminar on Application for Technology of Information and Communication (ISemantic). Piscatawy, NJ: IEEE, 2016, pp. 217-222.

[73] Y. Moffett, J. Dingel, and A. Beaulieu, "Verifying Protocol Conformance Using Software Model Checking for the Model-Driven Development of Embedded Systems," IEEE Transactions on Software Engineering, vol. 39, pp. 1307-13256, 2013.

[74] Mathworks. (2017). Chart Programming. Available: https://cn.mathworks.com/help/stateflow/programming-in-stateflow.html

[75] P. Muntean, A. Rabbi, A. Ibing, and C. Eckert, "Automated Detection of Information Flow Vulnerabilities in UML State Charts and C Code," in 2015 IEEE International Conference on Software Quality, Reliability and Security - Companion. Piscataway, NJ: IEEE, 2015, pp. 128-137.

[76] C. Wang, J. Wu, and H. Tan, "Revised Singleton Failures Equivalence for Labelled Transition Systems," Chinese Journal of Electronics, vol. 24, pp. 498-501, 2015.

[77] J. Tretmans, "Model Based Testing with Labelled Transition Systems," in Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected Papers, R. M. Hierons, J. P. Bowen, and M. Harman, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1-38.

[78] V. Valero, G. Díaz, and M. E. Cambronero, "Timed Automata Modeling and Verification for Publish-Subscribe Structures Using Distributed Resources," IEEE Transactions on Software Engineering, vol. 43, pp. 76-99, 2017.

[79] B. Marre, "LOFT: a tool for assisting selection of test data sets from algebraic specifications," presented at the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development, Aarhus, Denmark, 1995.

[80] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, "LUSTRE: a declarative language for real-time programming," presented at the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, Munich, West Germany, 1987.

[81] G. Shi, Y. Gan, S. Shang, S. Wang, Y. Dong, and P. C. Yew, "A Formally Verified Sequentializer for Lustre-Like Concurrent Synchronous Data-Flow Programs," in 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C). Piscataway, NJ: IEEE, 2017, pp. 109-111.

[82] W. Zheng, C. Liang, R. Wang, and W. Kong, "Automated Test Approach Based on All Paths Covered Optimal Algorithm and Sequence Priority Selected Algorithm," IEEE Transactions on Intelligent Transportation Systems, vol. 15, pp. 2551-2560, 2014.

[83] D. Angeletti, E. Giunchiglia, M. Narizzano, A. Puddu, and S. Sabina, "Using Bounded Model Checking for Coverage Analysis of Safety-Critical Software in an Industrial Setting," Journal of Automated Reasoning, vol. 45, pp. 397-414, 2010.

[84] N. Li and J. Offutt, "Test oracle strategies for model-based testing," IEEE Transactions

on Software Engineering, vol. 43, pp. 372–395, 2016.

[85] A. Hartman and K. Nagin, "The AGEDIS Tools for Model Based Testing," in UML Modeling Languages and Applications: <<UML>> 2004 Satellite Activities, Lisbon, Portugal, October 11-15, 2004, Revised Selected Papers, N. Jardim Nunes, B. Selic, A. Rodrigues da Silva, and A. Toval Alvarez, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 277-280.

[86] C. Rutz and J. Schmaltz, "An Experience Report on an Industrial Case-Study about Timed Model-Based Testing with UPPAAL-TRON," presented at the Fourth International Conference on Software Testing, Verification and Validation Workshops, Berlin, Germany, 2011.

[87] IEEE, IEEE Std 1474.4-2011: IEEE Recommended Practice for Functional Testing of a Communications-Based Train Control (CBTC) System. New York: IEEE, 2011.

[88] IEEE, IEEE Standard for Communications-Based Train Control (CBTC) Performance and Functional Requirements. New York: IEEE, 2004.

[89] M. Aguado, C. Pinedo, I. Lopez, I. Ugalde, C. D. L. Muñecas, L. Rodriguez, et al., "Towards zero on-site testing: Advanced traffic management & control systems simulation framework including communication KPIs and response to failure events," presented at the 2014 IEEE 6th International Symposium on Wireless Vehicular Communications (WiVeC 2014), Vancouver, BC, Canada, 2014.

[90] UNISIG, Functional Requirements for an on board Reference Test Facility

(Subset-094-0). ERTMS, 2009.

[91] J. Cullyer and W. Wai, "Application of formal methods to railway signalling-a case study," Computing & Control Engineering Journal, vol. 4, pp. 15-22, 1993.

[92] A. Piccolo, V. Galdi, F. Senesi, and R. Malangone, "Use of formal languages to represent the ERTMS/ETCS system requirements specifications," in 2015 International Conference on Electrical Systems for Aircraft, Railway, Ship Propulsion and Road Vehicles (ESARS), Piscataway, NJ: IEEE, 2015, pp. 1-5.

[93] S. Ghosh, A. Das, N. Basak, P. Dasgupta, and A. Katiyar, "Formal Methods for Validation and Test Point Prioritization in Railway Signaling Logic," IEEE Transactions on Intelligent Transportation Systems, vol. 18, pp. 678-689, 2017.

[94] M. Chai, L. Jidong, L. Hongjie, and Z. Lu, "Towards safety monitoring of ETCS level 2 with parametrized extended live sequence charts," in 2016 IEEE International Conference on Intelligent Rail Transportation (ICIRT), 2016, pp. 440-446.

[95] S. Li, X. Chen, Y. Wang, and M. Sun, "A Framework for Off-Line Conformance Testing of Timed Connectors," presented at the International Symposium on Theoretical Aspects of Software Engineering, Nanjing, China, 2015.

[96] J. Lv, K. Li, G. Wei, T. Tang, C. Li, and W. Zhao, "Model-based test cases generation for onboard system," 2013 IEEE Eleventh International Symposium on Autonomous Decentralized Systems (ISADS), Mexico City, Mexico, March 6–8 2013. Los Alamitos, CA: IEEE Chemical Society, 2013, pp. 1–6.

[97] M. Mikucionis, K. G. Larsen, and B. Nielsen, BRICS Report Series. RS-03-49: Online on-the-fly testing of real-time systems. Aarhus, Denmark: BRICS, 2003, p. 14.

[98] Z. Xiaolin, L. Teng, L. Kaicheng, and L. Jidong, "Online Testing of Real-time Performance in High-speed Train Control System," presented at the IEEE 17th International Conference on Intelligent Transportation Systems (ITSC), Qingdao, China, 2014.

[99] M. Broy, B. Jonsson, J.-P. Katoen, and M. Leucker, Model-Based Testing of Reactive Systems. Berlin: Springer, 1973.

[100] J. S. Keranen and T. D. Raty, "Model-based testing of embedded systems in hardware in the loop environment," IET Software, vol. 6, pp. 364 - 376, 2012.

[101] G. Gay, S. Rayadurgam, and M. Heimdahl, "Automated steering of model-based test oracles to admit real program behaviors," IEEE Transactions on Software Engineering, vol. 43, pp. 531–555, 2017.

[102] S. Hellebrand, H. J. Wunderlich, A. A. Ivaniuk, Y. V. Klimets, and V. N. Yarmolik, "Efficient online and offline testing of embedded DRAMs," IEEE Transactions on Computers, vol. 51, pp. 801-809, 2002.

[103] A. C. Dias-Neto and G. H. Travassos, "Supporting the combined selection of model-based testing techniques," IEEE Transactions on Software Engineering, vol. 40, pp. 1025–1041, 2014.

[104] A. David, K. G. Larsen, S. Li, M. Mikucionis, and B. Nielsen, "Testing Real-Time

Systems under Uncertainty," in Formal Methods for Components and Objects: 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers, B. K. Aichernig, F. S. de Boer, and M. M. Bonsangue, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 352-371.

[105] J. Tretmans, "Model based testing with labelled transition systems," in Formal Methods and Testing, R. M. Hierons, J. P. Bowen, and M. Harman, Eds. Berlin: Springer-Verlag, 2008, pp. 1–38.

[106] A. Guignard, J. M. Faure, and G. Faraut, "Model-based testing of PLC programs with appropriate conformance relations," IEEE Transactions on Industrial Informatics, vol. 14, pp. 350–359, 2018.

[107] K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou, "Testing Real-Time Embedded Software using UPPAAL-TRON: An Industrial Case Study," presented at the ACM International Conference On Embedded Software, Jersey City, NJ, USA, 2005.

[108] D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager, "Timed I/O automata: a mathematical framework for modeling and analyzing real-time systems," in 24th IEEE Real-Time Systems Symposium (RTSS 2003). Los Alamitos, CA: IEEE Chemical Society, 2003, pp. 166–177.

[109] R. M. Keller, "Formal verification of parallel programs," Commun. ACM, vol. 19, pp. 371-384, 1976.

[110] M. Mikucionis and E. Sasnauskaite, On-the-Fly Testing Using UPPAAL. Master's

thesis, Department of Computer Science, Aalborg University, Denmark, 2003.

[111] N. A. Lynch and M. R. Tuttle, "Hierarchical correctness proofs for distributed algorithms," presented at the Proceedings of the sixth annual ACM Symposium on Principles of distributed computing, Vancouver, British Columbia, Canada, 1987.

[112] D. Xu, M. Kent, L. Thomas, T. Mouelhi, and Y. L. Traon, "Automated Model-Based Testing of Role-Based Access Control Using Predicate/Transition Nets," IEEE Transactions on Computers, vol. 64, pp. 2490-2505, 2015.

[113] J. Tretmans, "Test Generation with Inputs, Outputs and Repetitive Quiescence," Software - Concepts and Tools, vol. 17, pp. 103-120, 1996.

[114] M. R. T. N.A. Lynch, "An introduction to input/output automata," CWI Quarterly, vol. 2, pp. 219–246, 1989.

[115] S. von Styp, H. Bohnenkamp, and J. Schmaltz, "A Conformance Testing Relation for Symbolic Timed Automata," in Formal Modeling and Analysis of Timed Systems: 8th International Conference, FORMATS 2010, Klosterneuburg, Austria, September 8-10, 2010. Proceedings, K. Chatterjee and T. A. Henzinger, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 243-255.

[116] R. Cardell-Oliver, "Conformance Tests for Real-Time Systems with Timed Automata Specifications," Formal Aspects of Computing, pp. 350-371, 2000.

[117] H. Ponce de León, S. Haar, and D. Longuet, "Conformance Relations for Labeled Event Structures," in Tests and Proofs: 6th International Conference, TAP 2012,

Prague, Czech Republic, May 31 – June 1, 2012. Proceedings, A. D. Brucker and J. Julliand, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 83-98.

[118] B. K. Aichernig and M. Tappler, "Symbolic Input-Output Conformance Checking for Model-Based Mutation Testing," Electronic Notes in Theoretical Computer Science, vol. 320, pp. 3-19, 2016.

[119] B. Beizer and J.Wiley, "Black box testing: Techniques for functional testing of software and systems," IEEE Software, vol. 13, p. 98, 1996.

[120] D. Giannakopoulou, C. S. Pasareanu, and C. Blundell, "Assume-guarantee testing for software components," IET Software, vol. 2, pp. 547-562, 2008.

[121] G. Behrmann, A. David, and K. G. Larsen, "A Tutorial on Uppaal," in Formal Methods for the Design of Real-Time Systems: International School on Formal Methods for the Design of Computer, Communication, and Software Systems, Bertinora, Italy, September 13-18, 2004, Revised Lectures, M. Bernardo and F. Corradini, Eds., ed Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 200-236.

[122] L. Dai, "Data for constructing experimental scenarios on testing the performance of rescheduling approaches," Data in brief, 2016.

[123] Taku Fujiyama, A. Chow, and B. Heydecker. (2017). DEDOTS: Developing and Evaluating Dynamic Optimisation for Train Control Systems. Available: http://www.ucl.ac.uk/railway-research/ongoing-projects/dedots

[124] F. Cassez, A. David, E. Fleury, K. G. Larsen, and D. Lime, "Efficient On-the-Fly

Algorithms for the Analysis of Timed Games," in CONCUR 2005 – Concurrency Theory: 16th International Conference, CONCUR 2005, San Francisco, CA, USA, August 23-26, 2005. Proceedings, M. Abadi and L. de Alfaro, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 66-80.

[125] "IEEE Draft Guide: Adoption of the Project Management Institute (PMI) Standard: A Guide to the Project Management Body of Knowledge (PMBOK Guide)-2008 (4th edition).IEEE P1490/D1, pp. 1-505, 2011.

[126] O. Tkachuk and M. B. Dwyer, "Environment generation for validating event-driven software using model checking," IET Software, vol. 4, pp. 194-209, 2010.

[127] P. Reales, M. Polo, J. L. Fernández-Alemán, A. Toval, and M. Piattini, "Mutation Testing," IEEE Software, vol. 31, pp. 30-35, 2014.

[128] R. Baker and I. Habli, "An Empirical Evaluation of Mutation Testing for Improving the Test Quality of Safety-Critical Software," IEEE Transactions on Software Engineering, vol. 39, pp. 787-805, 11 September 2012 2013.

[129] H. Roeahm, J. Oehlerking, M. Woehrle, and M. Althoff, "Reachset Conformance Testing of Hybrid Automata," presented at the HSCC'16 Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control, Vienna, Austria, 2016.

[130] J. H. Kim, K. G. Larsen, B. Nielsen, M. Mikučionis, and P. Olsen, "Formal Analysis and Testing of Real-Time Automotive Systems Using UPPAAL Tools," in Formal

Methods for Industrial Critical Systems: 20th International Workshop, FMICS 2015 Oslo, Norway, June 22-23, 2015 Proceedings, M. Núñez and M. Güdemann, Eds., ed Cham: Springer International Publishing, pp. 47-61, 2015.