An explanation of any relevant background you have in category theory (CT) or an
y of the specific projects areas:

I've first encountered CT when I started learning Haskell and Functional program
ming paradigm back in 2013. Then, during my masters degree, I used CT as a forma
l specification for my proposed framework which was about composing formal metho
ds with property-based testing in software engineering and I've published two pa
pers on this topic 1, 2. The framework was successful that I was offered a 3-mon
th contract to do research in one of the prestigious computer science organisati
on in Australia (data61). CT became a hammer for every problem (nail) I encounte
r in computer science and in life in general. Now, I'm a second year PhD student
 studying Blockchain (authenticated data structures) and I'm applying CT to desi
gn and implement my proposed framework (~unpublished~ categorical_ADS.pdf). I'm
also looking into consensus protocols inspired by Quantum Mechanics, so the cate
gorical-QM papers by Bob Coecke and Samson Abramsky are central to this part of
my research. Last but not least, I've been tweeting about CT and John Carlos Bae
z have liked one of them :-)

The date you completed or expect to complete your PhD and a
one-sentence summary of its subject matter.

I'll be completing my PhD by 2021 and my thesis is about improving blockchain at
 the level of authenticated data structures, as well as the consensus protocols
in blockchain networks inspired by Quantum computing

Order of project preference

*Simplifying quantum circuits using the ZX-calculus
*Toward a mathematical foundation for autopoiesis
*Traversal optics and profunctors
*Formal and experimental methods to reason about dialogue and discourse using ca
tegorical models of vector spaces
*Complexity classes, computation, and Turing categories
*Partial evaluations, the bar construction, and second-order stochastic dominanc
e

To what extent can you commit to coming to Oxford (availability of funding is un
certain at this time)

I'm able to attend for the whole two weeks even if there is no funding as I'm a
very interested and sponsored student.

A brief statement (~300 words) on why you are interested in the ACT2019 School.
Some prompts:

how can this school contribute to your research goals?

Currently, I'm trying to use ZX calculus to design quantum algorithms and protoc
ols for blockchains. I've got some ideas that I would like to discuss with the p
articipants. For instance, I think that we can combine the categorial realizatio
n of QM to design and implement blockchain consensus protocols. More specificall
y, since QM is a non-local theory and consensus protocols are non-local by defin
ition, I was thinking to use the non-locality proof as a vehicle to design my al

gorithms. I found some evidence that is pointing to that realization 3. Another idea is using QM features such as complementarity or Teleportation protocol for the same purpose

---

how can this school help in your career?

I’m a good fit for this opportunity as I did a lot of QM during my bachelor degree, then I did my masters in information technology, and now PhD in computer science. CT has allowed me to tame these subjects under simple yet powerful abstractions. Spending two weeks with top scientists in the field I’m interested in will improve my understanding and my knowledge which will definitely help in my career as a researcher in so many ways. I’m thinking to join quantum centre straight after I finish my PhD and I hope that attending this conf/school will open up collaboration opportunities in the future.

Nasser Ali Alzahrani

Software and Radiation Expert

✍ *King Fahad Qr, Riyadh, Saudi Arabia*

☎ *+966 569299341*

✉ nassersala@gmail.com

ϟ github.com/nassersala

Work experience

November 2016 – March 2017

Data61, Melbourne, Australia

Scholar

I was contracted by Data61 to research and work on Web-based curriculum visualisation.

February 2016 – November 2016

RMIT university, Melbourne, Australia

Research

We want software engineers to incorporate more Mathematics in their development process in order to develop correct systems. Therefore, my research involves the development of human oriented framework to reduce the gap between software engineers and formal methods (TLA+, FocusST and BeSpaceD).
I have published a paper on the subject in 3rd International Workshop on Human-Oriented Formal Methods (HOFM 2016), Springer

July 2008 – Present

iDevCode software, Riyadh, Saudi Arabia

Founder, Management and development

Software development for clients: development and configuration of backend in Amazon AWS and Google App-Engine, mobile applications for iOS and Android platforms, web applications and sites with Ruby on Rails framework. Development of Haskell backend API's. Development reactive and scalable systems based on ReactiveX paradigm.
We are embracing the server-less architecture for all upcoming projects as we believe this shift in computing will be the future for most cloud-based applications. Therefore we are investing in Amazon API Gateway, a service for creating and managing APIs, and AWS Lambda, a service for running arbitrary code functions, which both can be used together to simplify the creation of robust server-less architecture.

January 2003 – 2011

Prince Sultan medical city, Riyadh, Saudi Arabia

Senior Radiology Technologist

Managing people is a skill that I attained after working for many years as a supervisor technologist in Riyadh Military Hospital. I've been responsible for angiography department tender evaluation for three consecutive years. I've managed to reduce the time this process used to take by one third. I've negotiated directly with different international companies. Such as GE, Siemens, Philips, EV3, Boston Sientific and many more. I have provided training for the following systems: Siemens Artis, Philips Allura 15/12 and GE PACS system.

Education

2015 – 2016 **Master of Information Technology with Distinction**
RMIT, Melbourne, Australia

2015 **Principles of Reactive Programming**
Coursera Course
cole Polytechnique, Paris, France

2014 **Principles of Functional Programming**
EdX Course
TU Delft, Delft, Netherlands

2012– 2014 **Bachelor of Medical Radiation Sciences**
RMIT, Melbourne, Australia

2001– 2003 **Diploma in Radiological Sciences.**
PSMMC, Riyadh, Saudi Arabia

Public projects

2016 **Paper (hofm2016.wordpress.com/)**
Spatio-temporal model for property based testing

2015 **Swift (github.com/apple/swift)**
contributed one commit so far

2013 **CBDD (github.com/nassersala/cbdd/)**
behaviour driven development for c

Software skills

Good level	C, Objective-C, Swift, Java, Scala, Ruby, Haskell, git, shell, Unix, Reactive programming (RxJava, RxAndroid etc), Object-Oriented programming, Functional programming, Cloud computing: Amazon AWS, Google App-Engine, ruby on rails, purescript, elm
Intermediate	TLA+, PHP, LaTeX, MySQL, Python, Go, javascript, Cycle.js, React, nodejs, laravel
Basic level	Idris, Cog, prolog, Miranda, ML, APL, Postgres

RMIT UNIVERSITY

School of Science

GPO Box 2476
Melbourne VIC 3001
Australia

Tel. +61 3 9925 2283
Fax +61 3 9925 2454

/www.rmit.edu.au/science

Melbourne 22 Jan. 2019

TO WHO IT MAY CONCERN

I am writing this letter in support of Nasser Ali Alzahrani's application for ACT 2019 school. I am currently Mr. Ali Alzahrani's supervisor at School of Science, RMIT University.

Nasser Ali Alzahrani (3297335) is currently enrolled under my supervision into our PhD program at School of Science (Computer Science). He commenced his program research in Feb 2018. During his master's degree, Nasser published a couple of papers and he was granted a scholarship at CSIRO (Data61). He used Category Theory to formalize the framework.

Currently, he is applying category theory to both formalize and implement the framework that he is working on. The framework will make it easy to work with Authenticated Data Structures and Blockchains. He is also researching the relationship between quantum protocols to consensus in blockchain settings.

Nasser is a good student and he will definitely have interesting and fruitful contributions to the school.

If you have any questions, please feel free to contact me (see below for contact information).

Sincerely yours,

Ibrahim Khalil, PhD
Associate Professor
Computer Science and Software Engineering, School of Science
RMIT University
GPO Box 2476, Melbourne VIC 3001
Phone: 61 3 99252879
Email: Ibrahim.khalil@rmit.edu.au

Temporal Models for Formal Analysis and Property-based Testing

A minor thesis submitted in partial fulfilment of the requirements for the degree of Masters of Applied Science (Information Technology)

Nasser Ali Alzahrani
School of Science
Science, Engineering, and Technology Portfolio,
Royal Melbourne Institute of Technology
Melbourne, Victoria, Australia

November 17, 2016

Declaration

This thesis contains work that has not been submitted previously, in whole or in part, for any other academic award and is solely my original research, except where acknowledged.

This work has been carried out since March 2016, under the supervision of Dr. Maria Spichkova and Dr. Jan Olaf Blech.

Nasser Ali Alzahrani
School of Science
Royal Melbourne Institute of Technology
November 17, 2016

Acknowledgements

I would first like to thank my supervisors Dr. Maria Spichkova and Dr. Jan Olaf Blech for their support during the research. Their advices were crucial in helping me make the right decisions about how to proceed with my thesis.

I must express my very profound gratitude to my family for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Contents

List of Figures

List of Tables

Abstract

This thesis presents a framework to apply temporal models for formal analysis and property-based testing (PBT). The aim of this work is to help software engineers to understand temporal models that are presented formally and to make use of the advantages of formal methods: the core time-based constructs of a formal method are schematically translated to the BeSpaceD extension of the Scala programming language. This allows us to have an executable Scala code that corresponds to the formal model, as well as to perform PBT of the models' functionality. To model temporal properties of the systems, in the current work we focus on two formal languages, TLA+ and FOCUSST. To support PBT within the framework, we provided a PBT formalisation using the category theory and the corresponding automatisation.

Chapter 1

Introduction

One of the challenges in software engineering is to develop correct software. The software should meet user requirements, its properties should satisfy the model corresponding to design objective and the implementation should pass all functional tests. For Specifying safety-critical systems, it is not enough to use human language such as English to specify them. Furthermore, it is not enough to use controlled languages and semiformal languages which are subsets of natural languages which restricts the grammar and vocabulary in order to reduce or eliminate ambiguity – the precise and easy-to-read formal specification is essential to ensure that the safety properties of the system really hold. Moreover, the software development process should include aspects of human factors engineering, to improve the quality of software and to deal with human factors in a systematic way, cf. Spichkova et al. [2015]. Human factor aspects includes the design of human-computer interface of the software, development process that are human-related and the automatisation of such processes. According to the Engineering Error Paradigm (Redmill and Rajan [1997]), humans are seen as a "components of the system" (almost equivalent to software and hardware components in the sense of operation with data and other components), which are the most unreliable in the system.

Software errors can be fatal. One of the widely cited accidents in safety-critical systems are the accidents involved massive radiation overdoses by the Therac-25 (a radiation therapy machine used in curing cancer) that lead to deaths and serious injuries of patients which received thousand times the normal dose of radiation (Miller [1987]; Leveson and Turner [1993]). The causes of these accidents were software failures as well as problems with the system interface. The error was improbable to reproduce because it required very specific sequence of commands in order to occur. The improbability of the sequence makes the error unlikely to be noticed with manual testing because it is almost impossible to think of all

combinations of commands and edge cases. Automatisation might solve this problem, but the challenge is to create an automatisation which is not only efficient but also easy-to-use, i.e., is human-oriented.

Software errors can cause wasting of resources (Patra [2007]; Charette [2005]). An estimate of one trillion US dollars was spent on IT hardware, software and services by governments around the world, and in many cases they might be prevented by having a more human-oriented development process and methods. As per statistics presented by Dhillon [2004], humans are responsible for 30% to 60% the total errors which directly or indirectly lead to the accidents, and in the case of aviation and traffic accidents, 80% to 90% of the errors were due to humans. Thus, it is necessary to have human factors engineering as a part of the software development process.

Rigorous reasoning is the only way to avoid subtle errors in algorithms, and it should be as simple as possible by making the underlying formalism simple tools (Lamport [1993]). Formal methods (FMs) refer to a class of mathematical techniques used in development of large scale complex systems. These techniques can result in high-quality systems that can be implemented on-time, within budgets and satisfy user requirements (Bowen and Hinchey [1995]).

The value of FMs in real systems has far reaching consequences. For instance, FMs help engineers get the code right by getting the design right in the first place. Secondly, FMs help engineers gain a better understanding of the design. Despite all advantages, FMs are not widely used in large-scale industrial software projects for many reasons (Zamansky et al. [2016]). One of the core obstacles is the lack of readability and usability. The syntax of FMs is often too complicated and unreadable for novices, which makes an impression that all the FMs require huge amount of training. There also is a prejudice that the return of investment is very minimal and only justified in critical systems such as medical devices, what is generally not true (Newcombe et al. [2015]).

Temporal aspects of safety-critical systems are crucial to verify and to test a system, as in most cases the system properties should be analysed in relation to the time and to the location. To analyse temporal phenomena, we have to specify the corresponding spatial, temporal and event semantics formally and in a human-oriented way. The goal of our work is to increase usability of the analysis (in the sense of verification and testing) of the temporal aspects on the base of the corresponding formal models. The FMs flavours used in this work are: Temporal logic of actions (TLA+), FOCUSSTand BeSpaceD. TLA and FOCUSSTare used to model the system where BeSpaceD is used in lower level where logical operator are needed

and as an extension to Scala programming language.

Property based testing (PBT) allows us to generate huge numbers of system operations (e.g API calls or external events) and permute these operations in ways that is difficult for humans to think of. These combinations are then used to verify the system under test according to the temporal specification. For example, to test Queue data structure in C programming language using libraries that support PBT such as QuickCheck (Claessen and Hughes [2011-05]). The approach taken is that of *Sate Machines Models*. That is, to generate tests cases, and these test cases are sequences of API calls, in addition, the state of the system is modelled. For each API call, state transition function is defined to indicate how the state model changes. After that, post conditions functions are written to compare the real API call to the state of the model as shown in Figure 1.1

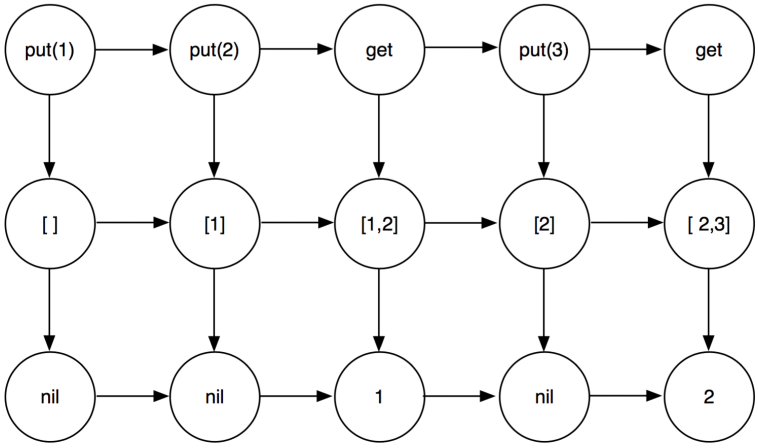


Figure 1.1: Testing Queue Example

The first row, represented by *put(1), put(2), get, put(3) and get()* respectively, corresponds to a single test case. To model the state of the queue buffer, a list of integers (that are supposed to be in the state) is used. Therefore, the model start with an empty list. then *put(1)* API call should correspond to the number *1* being present in the state (second row). The API call *put(2)* should append the number *2* to the list, and so on. The post conditions will then check the expected result (third row). In this case, the post condition checks that *get()* actually returns the *head* of the list. It is worth noting that the programming language that is used to write the API calls and the state model in this particular example is not in C language but in languages that support this kind of PBT such as Haskell programming language and Scala programming language to name few.

Figure 1.2 depicts the proposed framework that will allow for combining FMs with PBT. The FM specification gets translated to host programming language (Scala in this case). These

specification gets formal verification depending on the flavour of FM being used. For example, in case of TLA+, the TLA+ model checker (TLC) is used to check the specification as shown in Figure 1.3. On the other hand, in case of FOCUSST, the theorem prover Isabelle/HOL via the framework 'FOCUS on Isabelle' is used to verify systems specification as shown in Figure 1.4, cf. Nipkow et al. [2002] and Spichkova et al. [2007].

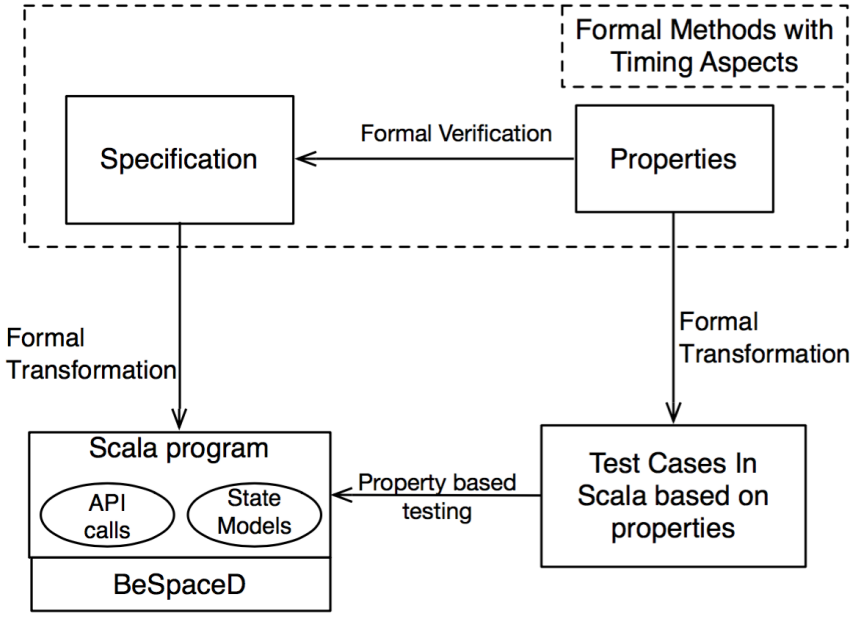


Figure 1.2: Proposed Framework

Figure 1.5 shows the testing aspect of the proposed model. At each test run, the testing aspect of the framework starts by generating API sequences of the system under test (SUT), creates state models from the specification formulas and then check the post condition function for compoarison. Each test will have instances of the API call sequences and the corresponding state model.

The implementation language of choice is Scala programming language. It was selected for the following reasons. First of all, it is one of the most popular languages on the Java virtual machine. The ecosystem will make it possible to find quick answers for questions that are related to technical aspects. Secondly, ScalaCheck is implemented in Scala. This will lower the impedance mismatch between research model and the host programming language. Finally, Scala, is a functional language. This will make working with the concepts of PBT more natural and simple. Haskell programming language is used for prototypical purposes due to its succinctness and richness. Swift programming language is also used as it makes interfacing with C code much easier especially when using swift version of QuickCheck.

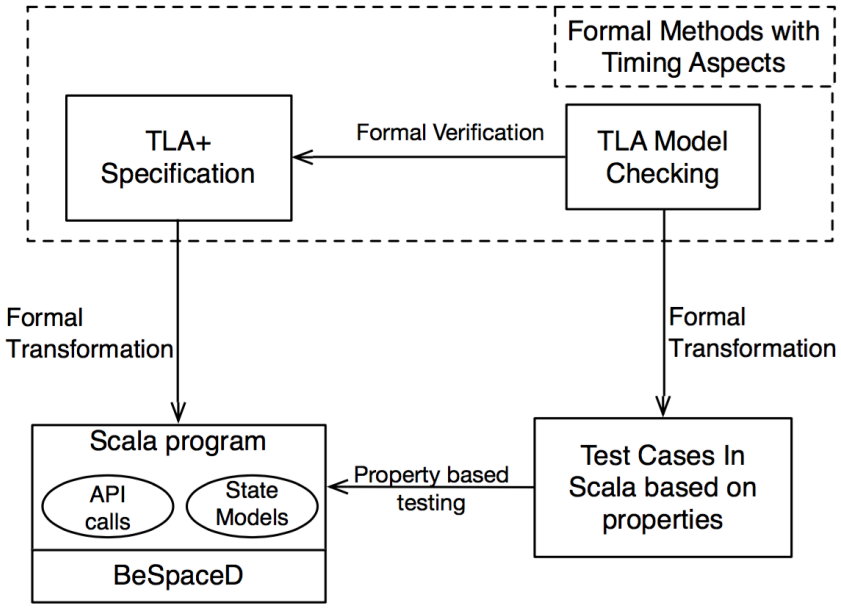


Figure 1.3: TLA+ Aspect of the framework

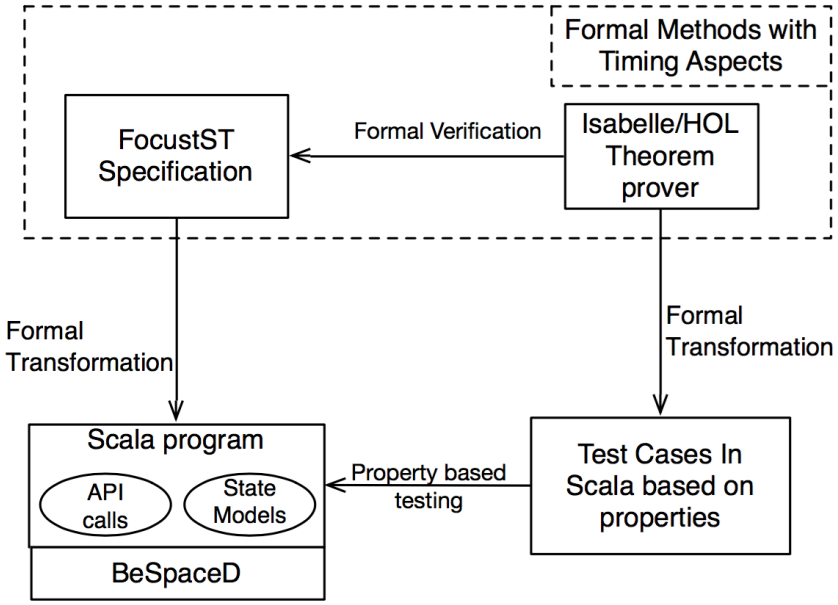


Figure 1.4: FOCUSST Aspect of the framework

For PBT, we apply and extension to *ScalaCheck library*. However, since the work proposes the substitution of the simplistic state machine in ScalaCheck with FMs, the use of this library is limited.

To relate the different modeling and abstraction layers to each other in the proposed frame-

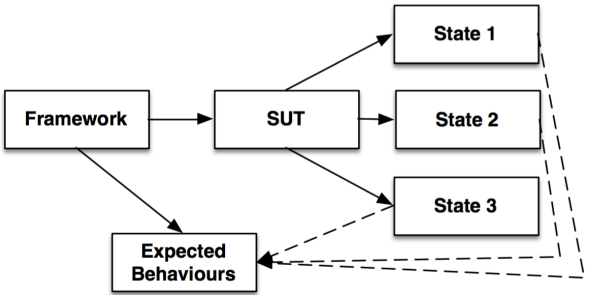


Figure 1.5: Testing aspect of the Proposed Framework

work, we are using category theory. Category theory helps in illuminating the relations of many aspects of the proposed ingredients that would be unseen otherwise. The categorical formalism relates the human actions (API call), system states (state model) and results to each other. Our FMs-based techniques will only be applied to the state model level. This will help to stair the direction of future investigation of the proposed model.

Contributions: The proposed framework will help to reduce the impedance mismatch between FMs representations and system code, which in turn will help in increasing the adoption rate by practitioners. Our framework aims at providing a set of application programming interfaces (APIs) to map programming language constructs to the FMs representation. This would allow the understandability of FMs to be improved indirectly, as the FM constructs will be expressed in terms of system code.

Preliminary work and core sections of this research has been published in the 3rd International Workshop on Human-Oriented Formal Methods, co-located with STAF conference 2016 in Vienna, Austria, cf. Alzahrani et al. [2016]

Chapter 2

Background

2.1 Formal Methods Models in Scala

Formal methods were introduced as a means of clearly specifying system requirements. Hinchey [2003] states that although formal methods are essential in the development of critical systems, they have not achieved the level of acceptance, nor level of use, that many believe they should. The uptake of formal methods has been far from ideal because many still believe that formal methods are difficult to use and require great mathematical expertise. Spichkova [2014] shows that in many cases simple changes of a specification method can make it more understandable and usable. It is possible to optimise the language without much effort. For example, adding constructs such as enumerations to formulas in complex specification makes its easier to validate and to be used as a discussion tool with experts. This optimisation is ignored most of the time due to it's obviousness.

Hinchey [2003] also states that in addition to the benefits of abstraction, clarification, and disambiguation, using formal methods at the formal specification level are invaluable documentation that greatly assist future system maintenance. This research incorporates specifications used in PBT to further help in precisely documenting the system. Lamport [1993] states two reasons for using formal methods formulas instead of programming language tailored to the specific problem:

- Specialized languages often have limited realms of applicability. A language that permits a simple specification for one system require a very complicated one for a different kind of system. The Duration Calculus seems to work well for real-time properties; but it cannot express simple liveness properties. A formalism like TLA+ that, with no built-in

primitives for real-time systems or procedures, can easily specify gas burner for example, it is not likely to have difficulty with a different kind of gas burner.

- Formalisms are easy to invent. However, practical methods must have a precise language and robust tools.

There are many examples where applying formal methods has lead to increasing reliability of systems. For example, a model checker TLC was developed for TLA formula was used to find errors in the cache coherence protocol for a Compaq multiprocessor, cf. Yu et al. [1999]. In addition, There are many other examples of successfully using formal methods to design systems, cf. Bowen and Hinchey [1995].

Despite the success stories for using formal methods in developing systems, the adoption rate is not ideal as mentioned before. Therefore, the proposed model will help in reducing the impedance mismatch between formal methods formulas and system code which in turn will help in increasing adoption rate by engineers. Those engineers might find expressing formal methods in languages they are familiar with to be less off-putting. As a result, adaptation rate by practitioners will be increased. It might be difficult to be come up with the specification formulas to design systems. Therefore, Usability section shows a technique that might help in that regard.

To create the initial set of formal methods-based modelling languages and tools, we have selected the following ingredients, which have a number of similarities in syntax and semantics and are also covering temporal aspects of the specifications:

- TLA+: Temporal logic of actions (TLA) is a logic developed by Leslie Lamport, which combines temporal logic with a logic of actions. It is used to describe behaviours of concurrent systems (Lamport [1994]).
- FOCUSST: Formal language providing concise but easily understandable specifications that is focused on timing and spatial aspects of the system behaviour (Spichkova et al. [2014; 2007]).
- BeSpaceD: A framework for modelling and checking behaviour of spatially distributed component systems (Blech [2015]; Blech and Schmidt [2014]).

Figure 2.1 shows user interface TLA Toolbox. The toolbox is an integrated development environment (IDE) for the TLA+ tools. It can can be used to create and edit specs where it shows any parsing errors after the spec file is saved. The tool box can also be used to

turn TLA+ model checker. The model checker needs some information from the model to check against for example, what properties to check and the values to substitute for constant parameters as shown in Figure 2.2. The Toolbox also provides is an error trace viewer and explorer that allows one to explore a structured view of the states, see exactly what parts of the state are changed in each step and evaluate arbitrary state and action formulas at each step in the trace. In addition, the toolbox also provide the ability to run the TLA+ proof system.

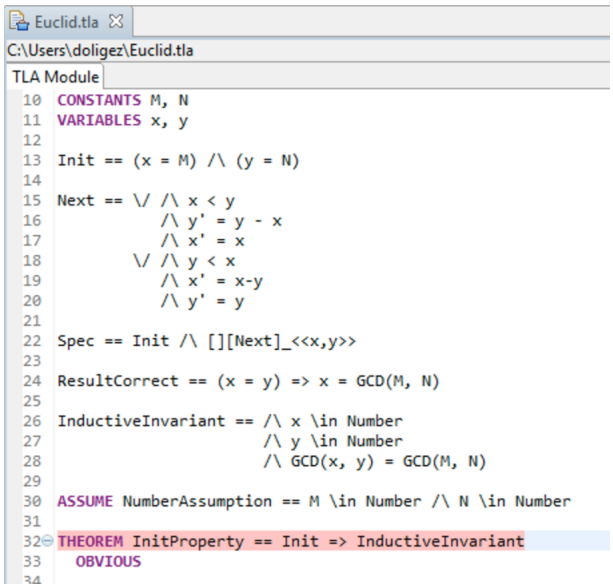


Figure 2.1: TLA Toolbox

The FOCUSST language was inspired by Focus, a framework for formal specification and development of interactive systems. In both languages, specifications are based on the notion of streams, cf. Broy and Stølen [2001b]. However, in the original Focus input and output streams of a component are mappings of natural numbers to single messages,whereas a FOCUSST stream is a mapping from natural numbers to lists of messages within the corresponding time intervals. The syntax of FOCUSST is particularly devoted to specify spatial (S) and timing (T) aspects in a comprehensible fashion, which is the reason to extend the name of the language by ST. The FOCUSST specification layout also differs from the original one: it is based on human factor analysis within formal methods (Spichkova [2012; 2013]).

FOCUSST is a modelling language that allows us to create concise and easily understandable specifications of safety-critical systems (SCSs). FOCUSST is also can be used for application of many specification and proof methodologies that allows writing specifications in a way that carrying out proofs is quite simple and scalable to practical problems. In particular, a speci-

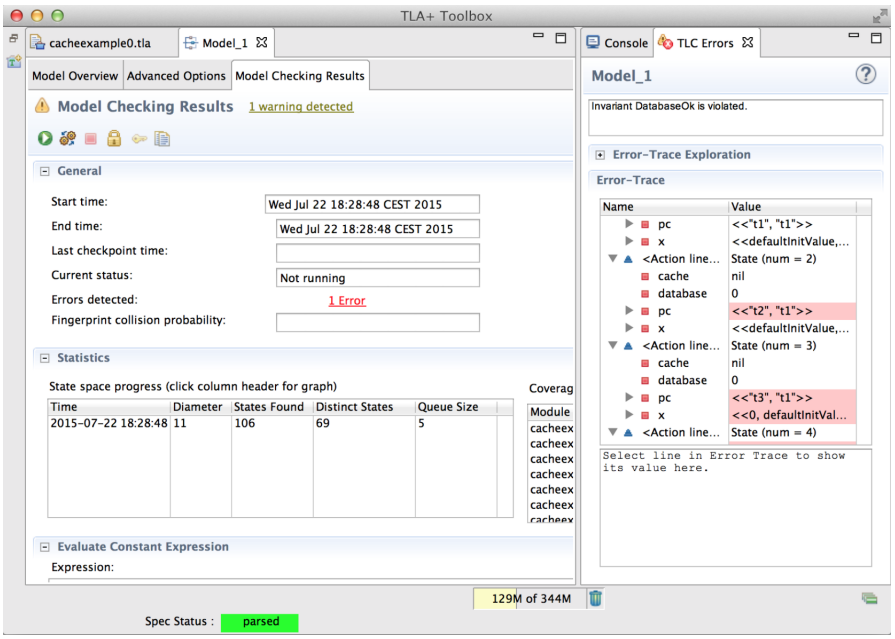


Figure 2.2: TLC Model Checker

fication of an SCS can be translated to a Higher-Order Logic and verified by the interactive semi-automatic theorem prover Isabelle. In addition, applying its component Sledgehammer. Sledgehammer employs resolution based first-order automatic theorem provers (ATPs) and satisfiability modulo theories (SMT) solvers to discharge goals arising in interactive proofs. Another advantage is a well-developed theory of composition as well as the representation of processes within a system.

FOCUSST supports real time and space requirements. Including time in the reasoning process to represent a real-time system makes the specification more readable. This simplifies the argumentation about its properties and gives a formal basis for verification. A suitable representation of SCSs should also make it possible to model information flow not only in time but also in space, because the spatial aspect may influence the delays of interactions between subcomponents of the system as well as between the system and the environment. This point is important for cost reduction of interoperability testing at the integration phase of the development process. For a versatile application of a notation, the selection of a suitable space-time coordinate system should be relatively free. The collection of FOCUSST operators over timing aspects and their properties can be specified and verified using the theorem prover Isabelle.

From a programming language perspective, we create BeSpaceD models by using Scala case classes. During the specification process, this gives a functional abstract datatype feeling with

a domain specific language flavour. A typical BeSpaceD formula is shown below

```
IMPLIES(AND(TimeInterval(300,605),Owner("AreaOfInterest")),
    OccupyBox(1051,3056,1505,3603))
```

The language constructs comprise basic logical operators (such as `AND` and `IMPLIES`). Furthermore special constructs for space, time, and topology are incorporated. In the example, `OccupyBox` represents a rectangular two-dimensional space while constructs such as `TimeInterval` allow for the modeling of temporal aspects possible. A variety of different operators exist which facilitates the reasoning about geometric and topological constraints. Furthermore, connections to data sources from cyber-physical systems exists (e.g., lego-trains Hordvik et al. [2016]) and event analysis for industrial automation facilities (Blech et al. [2015]) which facilitates the construction of demonstrators and conduction of experiments.

Design goals of BeSpaceD include:

- Ability to model spatial behaviour in a component oriented, simple and intuitive way
- Automatically analyse and verify systems and integration possibilities with other modelling and verification tools.

Blech and Schmidt proposed a process for checking properties of models and described the approach using different examples (Blech and Schmidt [2014]). In our current work, we only focus on the temporal constructs of BeSpaceD.

In our work we are using FOCUSST and TLA+ for modelling the behaviour of systems, whereas the BeSpaceD functionality is invoked at a lower level to check and test properties of the specified systems.

The workflow of the proposed framework includes the following:

- Informal specification of the system is created from systems requirements
- The informal specification is then transformed to formal specification, either in TLA+ or FOCUSST
- These TLA+ or FOCUSSTspecification is then checked formally in TLA+ model checker or Isabelle/HOL theorem prover, respectively
- The formal specification is then translated to Scala using the provided translation schema.

- The translated Scala specification is then fed to the extended ScalaCheck library
- The extended ScalaCheck part of the framework will check against the behaviour generated by FM specification instead of its default one.

To illustrate the wokflow, we use the example of Therac25 mentioned in the introduction. The machine included VT-100 terminal which controlled the PDP-11 computer. The sequence of user actions leading to the accidents was as follows:

- user selects 25 MeV photon mode
- user enters "cursor up"
- user select 25 MeV Electron mode
- previous commands have to take place in eight seconds

Therefore, we use algebraic data types to model the operations of the machine. Then we provide formal specification formulas and feed them to the framework as shown bellow:

```
sealed abstract class Operation
case object CursorUp extends Operation
case object Select25MevPhotonMode extends Operation
case object Select25MevElectronMode extends Operation
case object OtherKindOfOperation extends Operation

type Therac25 = Sut

val init: TLAInit =  {.. some predicate ...}
val next: TLANext = {.. another predicate ...}

val correctBehaviours: List[TLAState] =
   Therac25.correctBehaviours(init, next)
Therac25.checkAgainst(correctBehaviours, randoms(Operation))
```

The framework would generate large number of *Operation* combinations *(CursorUp, Select25MevPhotonMode, Select25MevElectronMode, and OtherKindOfOperation)* that are more likely to catch the error that caused the fatal accidents. Frequencies of generated commands can be tailored to match real system behaviour. The example used TLA+ formulas. However, FOCUSST formulas could have been used instead to specify the system.

2.2 Property-Based Testing

There are many kinds of software testing. One popular kind is that of *example based testing*. In this style, test cases requires one to provide an example scenario for each feature. That is, each example may exercise one feature of the system under test and the test runs only once with relevant input. Example based testing, is not enough to design critical system as engineers have to think about common case examples to include them in their test. They are very likely to overlook most combinatorial software errors.

Dually, *PBT* allows for the use of randomly generated tests based on system properties to test systems against their specifications and one test can run hundreds of times with different input values. An example of such library in Haskell programming language is *QuickCheck* (Hughes [2010]). Hughes (inventor of *QuickCheck*) showed that using this library allowed him to discover hundreds of bugs in critical systems such as automobiles and the DropBox file sharing service (Claessen and Hughes [2011-05]). However, *QuickCheck* uses Haskell programming language specific constructs (such as arrays, integers) and more complicated data types (such as algebraic data types) to model the specification of a system. Therefore, the work in this research proposes a foundation to have formal models (TLA+ or FOCUSST formulas) as specifications instead of Haskell constructs, as well as applicability of this approach for PBT of real systems.

Hughes states that Dijkstra was wrong when he claimed that testing can never demonstrate the absence of bugs in software, only their presence (Hu et al. [2015]), Hughes argues that if we test properties that completely specify a function (such as the properties of reversing a list) then PBT will eventually find every possible bug. In practice this is not true, since we usually do not have a complete specification, but this style of testing is very effective in exploring scenarios that no human can think of trying.

QuickCheck started as a testing framework for testing pure functional programs (Claessen and Hughes [2011-05]). However, recent development in the area of PBT incorporates the state-fulness of systems. That allowed for the testing of state-ful systems and even test programs written in imperative languages such as C programming language (Gerdes et al. [2015]; Hughes [2010]). Hughes states that testing state-ful systems is challenging. He argues that the state is an implicit argument to and result from every API call, yet it is not directly accessible to the test code. Therefore, his solution was to model the state abstractly and introduce state transition function that model the operations in API under test.

However, the state transition in *QuickCheck* is modelled manually using *pre*, *post* and *next*

functions for every operation in the system under test. On the other hand, our framework will generate these transitions automatically using specification formulas.

Adaptive Random Testing (ART) is a way of increasing diversity in random tests (Chen et al. [2010]). Rather than running every random test generated, ART repeatedly generates a set of candidate tests, but only runs the one 'furthest' from any previous run test. This leads to a variety of tests, and has been shown to reduce the mean number of test to reduce the number of bugs by up to 5 per cent compared to ordinary PBT. However, it requires a distance metric to be included on test cases, which is not easy for non-numerical software, and the overheads of the method can make it slower than random testing in practice (Arcuri and Briand [2011]).

Figure 2.3 depicts PBT aspect of the proposed framework that will allow for combining FMs with PBT. The first row (API calls) represents the actual system under test. The second row represents the world in which the specification formulas lives. The Behaviour between subsequent API calls is modelled through a function of discreet time. Time functions are mapped to the corresponding state transitions between states. The general idea is to start with specifying the system using human-oriented modelling techniques based on FMs. After the specification phase, the software of the system under test is designed according to the specification. The Framework will then generate random test cases to exercise and verify that the system runs according to the specification. If a test fails, it will be the judgment of the engineer to decide whether the errors were in the system software or in the specification formulas for which the system was not correctly specified. If the test passes without any errors, the system under test meets the specification.

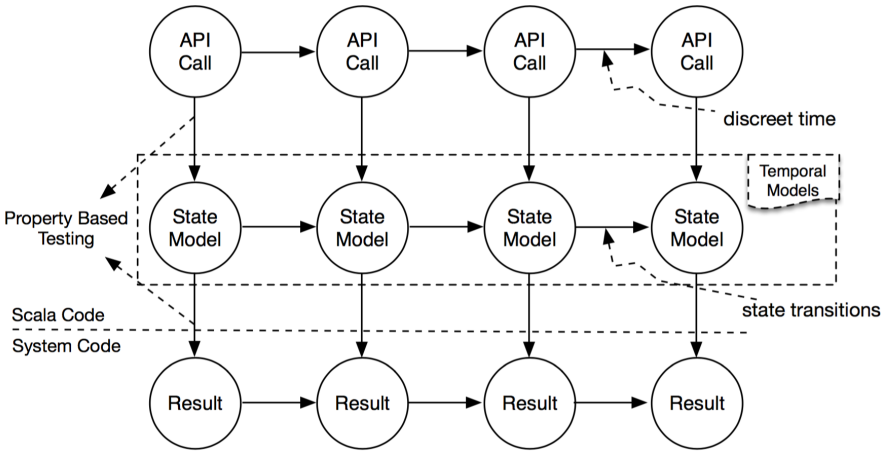


Figure 2.3: Property-based Testing in The Proposed Framework

2.3 Category Theory

Category Theory (Rydeheard and Burstall [1988]) is used to give formalism to the proposed framework. In this section we give brief introduction to *Category Theory.* Next we give the categorical formalism of the proposed framework.

Category theory is a branch of mathematics and is concerned to study algebraic structures in an abstract approach. Goguen [1991] states that category theory can be beneficial for computer scientists for the following reasons:

- Formulating definitions and theories. In fields that are not yet very well developed like computer science, it is often difficult to formulate basic concept of research. Category theory provide relatively explicit measures of elegance and coherence that can be helpful in this regard.
- Carrying out proofs. Once basic concepts have been correctly formulated in a categorical language, *Diagram chasing* ease the process of proofing properties in a natural way.
- Discovering and exploiting relations with other fields, Finding similar structures in different areas suggests trying to find further similarities. For example, an analogy between *Petri nets* and the *lambda calculus* might suggest looking for a closed category structure on nets which seems to open an entirely new approach to concurrency.
- Formulating conjectures and research directions. For example, if a functor was found, its adjoints can be similarly investigated.
- Dealing abstraction and representing independence. In computer science, abstract viewpoints are often better, because of the need to achieve independence from complex details of how things are modelled or implemented. As a consequence of the first guideline is that two objects can be considered "the same" and treated in a similar fashion if and only if they are isomorphic to each other.

A category consists of objects and morphisms (arrows). Every morphism has a domain and codomain. A morphism $f : A \to B$ has object A as its domain and B as its codomain. For every category, if there are morphisms $f : A \to B$ and $g : B \to C$, then there exists a composite morphism $g \circ f : A \to C$. Composition must be associative. That is, $(h \circ g) \circ f = h \circ (g \circ f)$. Every object X has an identity morphism Id_X that map the object to itself. For every morphism $f : A \to B, Id_B \circ f = f = f \circ Id_A$.

Functors are structure preserving morphisms between categories. A functor $F : C \to D$ maps each object of category C into a corresponding object of category D, and maps each morphism of category C onto a corresponding morphism of category D. Natural Transformation are mapping between Functors. For example, For two functors F, G: $\mathcal{C} \to \mathcal{D}$ between categories $\mathcal{C}$ and $\mathcal{D}$, a family of arrows $\alpha A : FA \to GA$ is a natural transformation if, for any $f : A \to B$, the following diagram commutes:

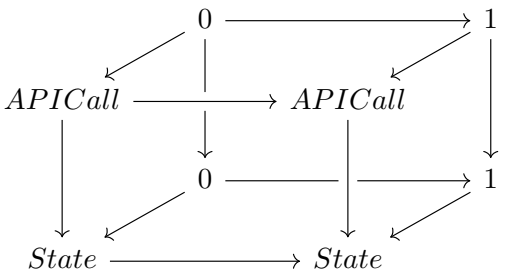
$$\begin{array}{ccc} FA & \xrightarrow{\alpha A} & GA \\ \downarrow{\scriptstyle Ff} & & \downarrow{\scriptstyle Gf} \\ FB & \xrightarrow{\alpha B} & GB \end{array}$$

Chapter 3

Proposed Framework

3.1 Category Theory Formalisation

In this section, we shall give a categorical formalism for the implemented. Firstly, we observe that for every time point there is a corresponding *API* call and a *system state*. This can be visualised through the following diagram:



The property based testing aspect of the framework can be thought of as *sets over time* (discreet time) which forms a category. The object of which are functors in the form of: $Sets^{\omega}$ where ω is a category of partially ordered set of the *natural numbers* $\mathbb{N}$, the object of which are natural numbers equiped with morphisms α which represent its order relation ($\leq$) as shown in the following diagram:

$$0 \xrightarrow{\alpha 0} 1 \xrightarrow{\alpha 1} 2 \xrightarrow{\alpha 2} 3 \xrightarrow{\alpha 3} 4$$

Therefore, the morphisms between objects (Functors) are *Natural Transformations*. In our framework, every row is an object in this category. The state model row (second row) is where the specification temporal models reside.

3.2 Application to Formal Methods

In this section, we show the applicability of the proposed framework to different flavours of FMs, in particular, FOCUSSTand TLA+, beSpaceD. The goal is to show that the proposed framework can be applied to many types of FMs with similar syntax. Each subsection presents systematic informal program transformation schemas. Using these schemas makes transforming FM formulas to any hosting language, Scala in this case, an easy mechanical task.

We start by analysing TLA+ syntax and semantics. After that, we show the design and model the API for the TLA+ flavour. After that, we show the designed API and the testing it using small example (*One Bit Block*). Similar process applied to FOCUSST, showing the analysis of FOCUSSTsyntax and semantics. Restricting FOCUSSTto it's major parts that is related to temporal properties. FOCUSSTsection includes the implementation of the designed API and testing it on another and example that involves rails track and self driving cars.

3.2.1 Application to TLA+

We start by analysing and describing TLA+ syntax and semantics. The following BNF Grammar is an excerpt from the full TLA+ specification which describe expressions of TLA+ grammar. Terms that begin with G, represent nonterminals. The terminals are sets of tokens, described with the operators tok and Tok. The complete BNF grammar for TLA+ can be found in *Specifying Systems* book, cf. Lamport [2002]

```
G.Expression ::=
         |  G.Expression & InfixOp & G.Expression
         |     tok("{")
           &  G.Expression
           &  tok(":")
           &  CommaList(G.QuantifierBound)
           &  tok("}")
        ...
        |  G.Expression & tok("[") & CommaList(G.Expression)
             & tok("]")
        ...
     |  Number
     |  String
     |  tok("@")
```

In addition, The TLA+ formulas grammar can be captured by the following grammar:

```
TLAFormula := TLAInit | TLANext
TLAInit :=  TLAVar == TLAValue
TLANext := TLAVar == TLAValue | TLAPrimed == TLAValue
```

A TLA formula such as $Init \wedge \Box[Next]v$ specifies the initial states and the allowed transitions of a system. It allows for transitions that do not change the value of v. This kind of transitions is called *stuttering transitions*. Most TLA system specifications are of the form $Init \wedge \Box[Next]_v \wedge L$. The semantics of such formulas are shown in Table 3.1.

Table 3.1: Semantics of TLA formula

Init	State formula describing the initial state(s)
Next	Action formula formalizing the transition relation – usually a disjunction $A1 \vee .. \vee An$ of possible actions (events) Ai
L	Temporal formula asserting liveness conditions

Table 3.2 shows logic operators in TLA+ and their mappings in Scala, many of the logical operators in Scala are provided by BeSpaceD.

To illustrate the description TLA+ we use the following example formula that specify a simple program that increment x or y repeatedly:

```
Init ==  (x = 0) /\ (y = 0)
M1 == (x' = x + 1) /\ (y' = y)
M2 == (y' = y + 1) /\ (x' = x)
M == M1 \/ M2
```

The TLA+ language has some notations and concepts that are not ordinary math. However, these notations are used mainly for advanced applications. Therefore, for the purpose of this research, a subset of TLA+ has been used. The subset includes variables which are flexible variables of temporal logic. The rigid variables of predicate logic is ignored. The class of values includes functions, numbers, strings and sets. A TLA formula is true or false of a behaviour. These formulas are built up from state functions using Boolean operators such as Not, And, Or, Implication and Equivalence ($\neg, \wedge, \vee, \Rightarrow, \equiv$), respectively. A primed variable in TLA+ represent to value of that variable in the next state relation. In TLA+, the state function is an expression built from variables, constants and constant operators. It assigns a value to each state. For example, *x + 1* assigns one plus the value that the state currently

Table 3.2: Operator mapping from TLA+ to Scala

TLA+	Scala
$/\backslash$	AND
$\backslash/$	OR
$\Rightarrow$	IMPLIES
TRUE	TRUE
FALSE	FALSE
BOOLEAN	Boolean
$\{TRUE, FALSE\}$	List(TRUE, FALSE)
$\leq$	lessThanEq
$\geq$	greaterThanEq
$>$	greaterThan
$<$	lessThan
$\not\leq$	lessThanEqNot
$\not<$	lessThanNot
$\not\geq$	greaterThanEqNot
$\not>$	greaterThanNot
$\in$	IN
$x == e$	defined(x, e)
$x = e$	assign(x, e)
$\forall x \in S : p$	for $\{x \leftarrow S$; if p$\}$ yield x
$\exists x \in S : p$	exists(x, S, p)
CHOOSE $x \in S$	choose(x, List(S))

have for x. An action is an expression containing primed and unprimed variables. It is either true or false of a pair of states. The primed variables referring to the second state. For instance, the action M is true for (s, t) if and only if the value that state t assigns to x equals one plus the value that state s assigns to x and the values assigned to y by state s and state t are equal. The pair of states that satisfy an action A is known as a *step*. Therefore, $M1$ step is one that increments x by one and does not change y. If f is a state predicate, f' means the expression obtained by priming all the variables of f. For instance, $(x + 1)'$ equals $x' + 1$, and $Init'$ equals $(x' = 0)/\ (y' = 0)$.

In TLA+, a representation of an abstraction of a system is modelled using the standard model. The Standard Model states that an abstract system is described as a collection of behaviors, each representing a possible execution of the system, where a behaviour is a sequence of states and a state is an assignment of values to variables. In this model, an event (step) is the transition from one state to the next in a behaviour.

For example, In one-bit clock, formulas are defined as follows:

```
VARIABLE b

 Init ==  (b = 0) \/ (b = 1)
 Next == \/ /\ b = 0
                 /\ b' = 1
              \/ /\ b = 1
                 /\ b' = 0
```

These two TLA+ statements define *Init* and *Next* to be the two formulas. Therefore, referencing *init* or *Next* is completely equivalent to typing ((b = 0) \/ (b = 1)). The equality symbol = (typed ==) is read *is defined to equal*.

To be able to transform these two formulas to host programming language, it is necessary to capture the essential aspects of the formula to be transformed. Such a capture is known as a schema. Each transformation step will consist of two schemata: one to capture the TLA+ formula and one to capture the corresponding programming language function. The two schemata together can then be used to do the transformation. Here is the TLA+ version of the schema:

```
f1 == p \/ q
f2 ==  \/ /\ p
          /\ q
       \/ /\ q
          /\ p
```

That is, f1 represent Init, f2 represent Next, p represent (b=0), q represent (b=1) respectively. Table 3.3 shows mappings between TLA formulas and Scala programming language for the *One Bit Clock* example.

Table 3.3: Mapping between TLA+ and Scala

Schema	TLA+	Scala
f1	Init	TLAInit
f2	Next	TLANext
p	b = 0	defined(b, 0)
q	b = 1	defined(b, 1)
?	Variable	TLAVariable
?	/\	AND
?	\/	OR

According to the schema, there translation of one bit clock from TLA+ to Scala is as follows:

```
val b: TLAVariable = TLAVariable(IN(List(0, 1)))

val init: TLAInit =  OR(defined(b,0), defined(b,1))

val next: TLANext = {
    while(true) {
        if defined(b, 0)
           return assign(b, 1)
        else
          return assign(b, 0)
     }
}
```

3.2.2 Application to FocusST

FocusST provides a theory covering real time and space requirements. This is essential to avoid errors that are a result of mistreating or excluding them. In addition, including time in the reasoning process to represent a real-time system makes the specification more readable compared to untimed one. However, for the purpose of this work, only the time aspect is considered.

The base data types we use in FocusST are shown in Table 3.4. The FocusST *records type RV* is defined also using the *Focus* rules, where $con_1, \ldots, con_n$ and $sel_1^1, \ldots, sel_{k_n}^n$ are constructors and selectors respectively:

type $RV = con_1(sel_1^1 \in T_1^1, \ldots, sel_{k_1}^1 \in T_{k_1}^1)$
...
$con_n(sel_1^n \in T_1^n, \ldots, sel_{k_n}^n \in T_{k_n}^n)$

Table 3.4: FOCUSST *Base Data Types*

Bool	the type of truth values
$\mathbb{N}$	the type of natural numbers
Bit	the type of bit values 0 and 1
type $T = e_1 \vert ... \vert e_n$	enumeration type
type $T = \{e_1, ..., e_n\}$	enumeration type
type $L = \mathbb{N}$	list type over $\mathbb{N}$
$\mathbb{N} \implies T^*$	Infinite timed streams of type T
$(T^*)^*$	Finite timed streams of type T are defined by list of lists overt this type, where T^* denotes a list of elements of type T

A template for a general FOCUSSTspecification is shown in Figure 3.1 The *in* and *out* sections are used to specify input and output streams of the corresponding types:

- $x_1, ..., x_m$ are input streams of the type $InType_1, ..., InType_m$, respectively
- $y_1, ...y_n$ are input streams of the type $OutType1, ..., OutType_n$, respectively.

local and *init* sections include local variables and initial values, respectively. $v_1, ..., v_k$ are local variables of the types $VarType_1, ..., VarType_k$

Specifying every component in FOCUSSTuses assumption-guarantee-structured templates. Because the specified component is required to fulfil the guarantee only if its environment behaves in accordance with the assumption, the assumption-guarantee-structured templates help in the avoidance of the omission of unnecessary assumptions about the system‘s environment.

The keyword *asm* lists the assumption that the specified component expect from its environment, for example that all the input streams should contain exactly one message per time interval. The component behaviour that should be guaranteed in the case all assumptions are fulfilled, is then described in the specification section *gar*. Each formula in the assumption and guarantee section is numbered. FOCUSSTproposes to number assumptions for assumptions, initial guarantees and core guaranteed behavioural properties as shown in Table 3.5. The behavioural properties are usually either defined over all time intervals $t \in$ N or are presented by the corresponding predicates, for example, *ts*.

Table 3.6 shows the mappings between FOCUSSTbasic operator to Scala. In addition, The predicate

Table 3.5: FOCUSST *Numbering of Specification Template*

Assumptions	$A1, A2, A3, ...$
initial guarantees	$I1, I2, I3, ...$
Behavioural properties	$B1, B2, B3, ...$

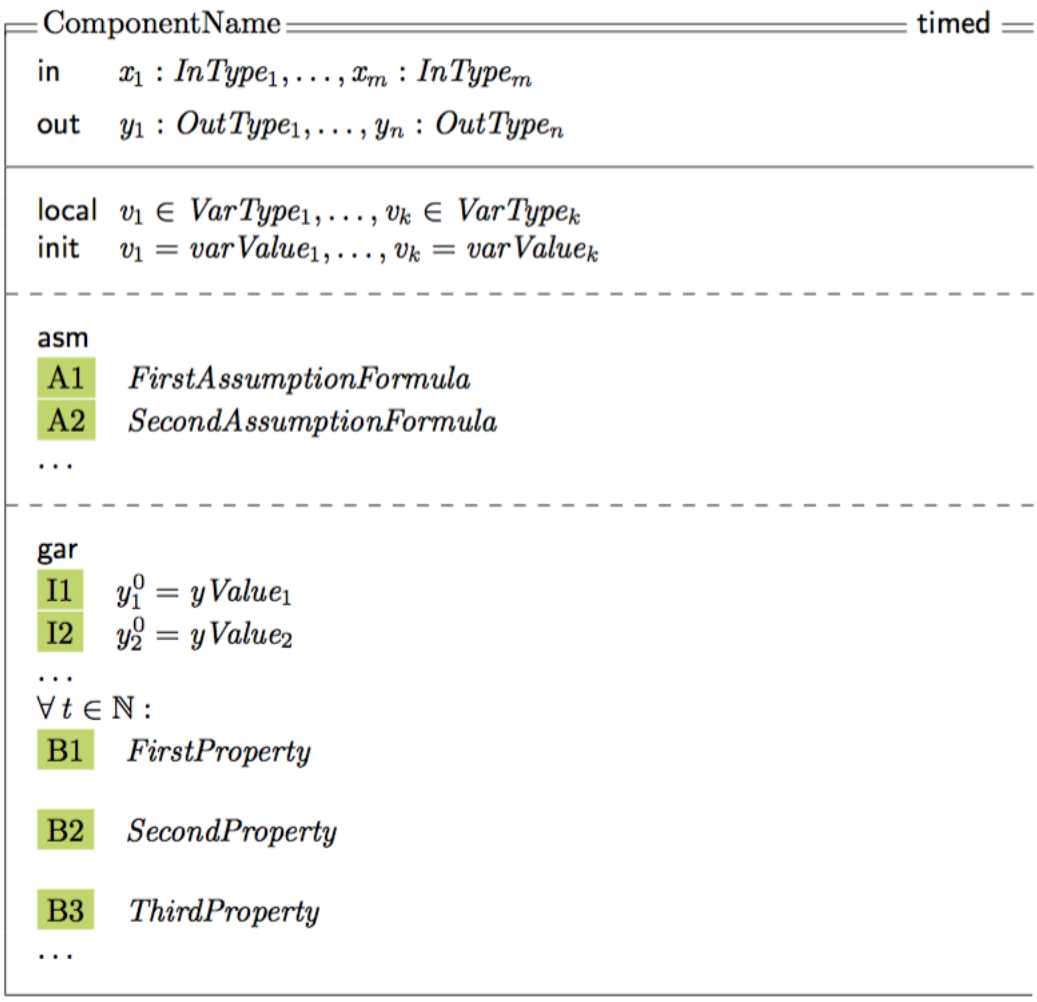


Figure 3.1: FOCUSST *Template (Spichkova [2016])*

$msg_s(k)$ holds if each time interval of the stream s has at most k elements. For example, $msg_s(1)$ indicates that each time interval of s either has a single element or is empty. The predicate $ts(s)$ ensures that each time interval of the stream s contains exactly one message. Therefore, $ts(s)$ implies $msg_s(1)$, however, the contrary is not true, that is, $msg_s(1)$ does not imply $ts(s)$.

The example that are we going to use to test the application of the proposed framework is that

Table 3.6: Operator mapping from FOCUSST *to Scala*

FOCUSST	Scala
$/\backslash$	AND
$\backslash/$	OR
$\rightarrow$	IMPLIES
TRUE	TRUE
FALSE	FALSE
BOOLEAN	Boolean
$\leq$	lessThanEq
$\geq$	greaterThanEq
$>$	greaterThan
$<$	lessThan
$\not\leq$	lessThanEqNot
$\not<$	lessThanNot
$\not\geq$	greaterThanEqNot
$\not>$	greaterThanNot
$\in$	IN
$x == e$	defined(x, e)
$x = e$	assign(x, e)
$\forall x \in S : p$	for $\{x \leftarrow S$; if p$\}$ yield x
$\exists x \in S : p$	exists(x, S, p)
$\langle\rangle$	List()
$\langle a_1, \ldots, a_m \rangle$	a_1 to a_m

of rails track with self driving cars. Fig 3.2 shows a train *RTrack* shuttles on a rail track that is crossed by three roads. On each road, an autonomous vehicle *AVehicle* is operating. The location, speed, movement direction and other attributes for the four mobile units are specified by number of constraints. While passing one of the critical points tr_i that are close to the crossings ($1 \leq i \leq 6$), the train sends a *wait* signal to the relevant *AVehicle* to avoid collisions. If the train is heading towards a critical point, time interval represented by the *wait* signal is only sent if the *AVehicle* has to stop. This calculated if *AVehicle* is heading towards the crossing and expected to stop in due time. The time interval may depend on the speed of the train giving it sufficient time to pass the crossing. When *wait* contains *0*, this indicates that the vehicle can keep moving. Generally speaking, It is difficult for the space-related behaviour to be accurately incorporated in the specification process because the train leaving the crossing has to be guaranteed. In addition, This behaviour should only take place when the time interval has passed.

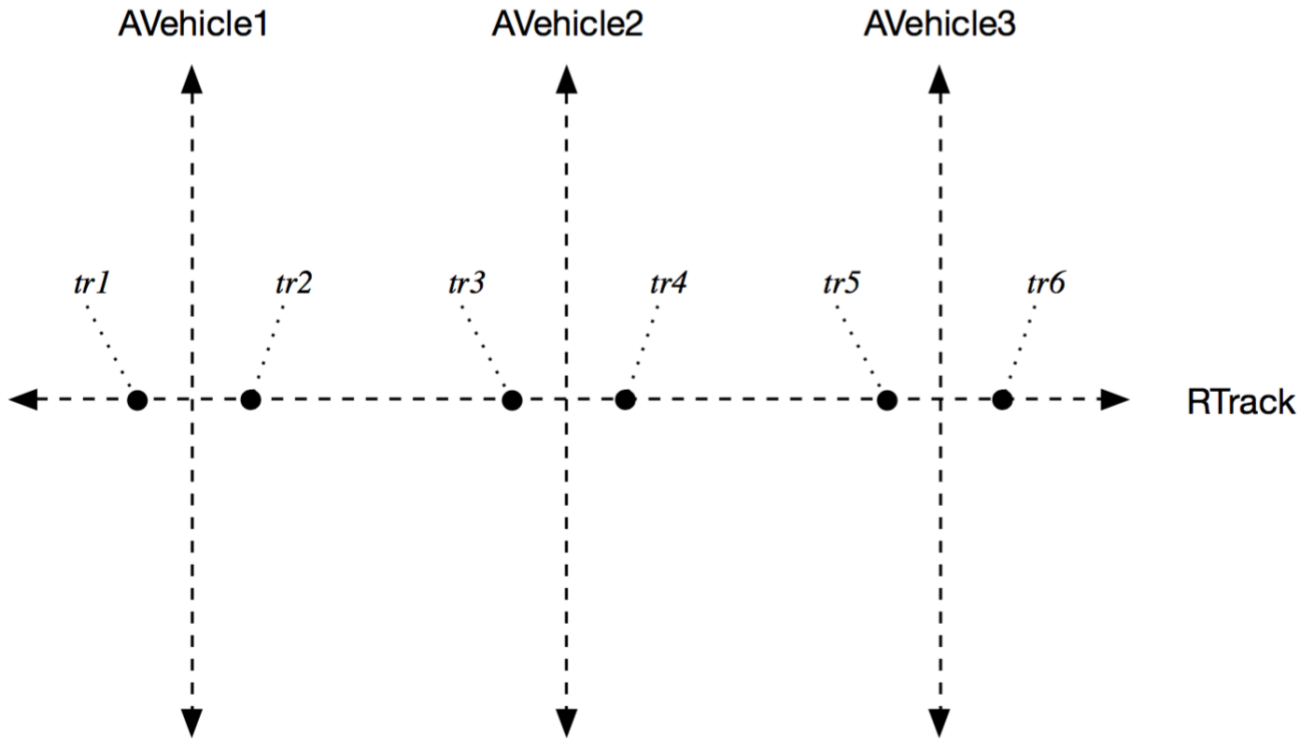


Figure 3.2: Automatic transport system

What follows is the *spObject* specification pattern for the component *AVehicle*. The interface part of the specification shows that *AVehicle* has delay of one time unit. It also shows *wait* and *tSpeed* of type $\mathbb{N}$ as well as *tDir* of type *Directions*. To notify changes of the target speed and the target direction of the object, the ports *tSpeed* and *tDir* are used. If *AVehicle* is too close to the crossing with *RTrack*, it is informed through the *wait* port to stop for some time. After that, it moves with the same previous speed. Label *out* defines the output channel *resp* of type *Event* that includes a single element event

used to signal the start of movement by the vehicle:

```
in wait, tSpeed : ℕ, tDir : Directions
out resp : Event
local timer, lspeed
asm msg1(wait) ∧ msg1(tSpeed) ∧ msg1(tDir)
gar reps0 = ⟨⟩
```

$\forall t \in \mathbb{N}:$

$$wait^t = \langle\rangle \wedge timer = 0 \rightarrow Upd(speed, tspeed^t) \wedge Upd(direction, tDir^t) \wedge Move()$$
$$wait^t \neq \langle\rangle \wedge ft.wait^t > 0 \wedge timer = 0 \rightarrow timer' = f.wait^t \wedge lspeed = speed \wedge speed' = 0$$
$$wait^t \neq \langle 0\rangle \wedge timer \neq 0 \rightarrow resp^{t+1} = \langle event\rangle \wedge timer' = 0 \wedge speed' = lspeed$$

Again, to be able to transform these two formulas to host programming language, we use similar transformation schema to that of TLA+ shown in the table 3.7. The table shows only a subset of the schema for brevity. The schema can be extended to accomodate the full example.

Table 3.7: Mapping between FOCUSST *and Scala for Rail Track Example*

FOCUSST	Scala
Init	FocusInit
Next	FocusNext
timer = 0	defined(timer, 0)
wait = $\langle\rangle$	defined(wait, List())
Variable	FocusVariable
/\	AND
\/	OR

The following is the Scala translation after applying the translation schema. Some auxiliary functions that is responsible for updating the state used. The translation excerpt omitted these auxiliary functions and some variables definitions for brevity:

```
val resp: FocusVariable = FocusVariable(List())
val timer: FocusVariable = FocusVariable(0)
val wait: FocusVariable = FocusVariable(List())

val init: FocusInit =  AND(resp, timer, wait)

val next: FocusNext = {
```

```
    while(pred) {
        if (And(defined(wait, EmptyFocusList(resp)),
                    defined(timer, 0))) {
           return AND(Update(speed, tSpeed),
                              Update(direction, tDir),
                              Move())
         } else if (And(defined(wait, NotEmptyFocusList(resp)),
                       defined(timer, 0)),
                       greaterThan(wait, 0)) {
                         return And(assign(timer, wait),
                                          assign(lspeed, speed), resetSpeed())
         } else { return And(defined(wait, FocusListContainsNot(0)),
                            FocusOutPut(FocusList(resp))
          }
     }
}
```

The evaluation section (5.1) contains a real world example based on Steam Boiler system. It shows the workflow of the proposed framework in more details.

Chapter 4

Developed Tools

In this chapter, we present the tools we developed to allow for the use of FM's to be integrated with PBT. Firstly, a subset of TLA+ model checker in Scala is presented. Section 4.2, contains information about the random generator of BeSpaceD constructs that is used in the phase of testing. Section 4.3 introduces the system behaviour generator that will allow the API permutations to be checked against. These permutations and their role in the proposed framework are explained in the Section (4.4). The chapter is concluded with a proposed method that help in the usability of the proposed framework.

4.1 A Subset of TLA+ Model Checker in Scala

A subset of TLA+ model checker was developed as a tool to generate the state of behaviour. The tool plays another important role in the framework; it verifies that the schematic translation from TLA+ syntax to Scala is correct. The correctness can be inferred by comparing the behaviour generated form TLA+ model checker with the behaviour generated by this tool. The process underwent the use of directed graph $\mathcal{G}$, the nodes of which are states. The algorithm is as follows:

- assign to $\mathcal{G}$ the set of all possible initial states of behaviours. The states are computed by permuting all possible assignments of values to variables that make the initial predicate true.
- Then, for every state s in t, compute all possible states t such that $s \rightarrow t$ is a valid step in the behaviour. This is done through the substitutions of the values assigned to variables by s for the unprimed variables in the next-state actions, and then compute all possible assignments of values to the primed variables that make the next-state actions true.
- Then, for every state t found, we first add t to $\mathcal{G}$ if it is not already there. Then, we draw an edge from s to t
- The process is repeated until no new states or edges can be added to $\mathcal{G}$

If this process terminates, the nodes of $\mathcal{G}$ consists of all reachable states of the specification. Every behaviour satisfying the specification can be found by starting in an initial state and following a path in $\mathcal{G}$

Following this approach, now we can test the system by permuting all APIs sequences and checking that every results sequence generated can indeed be found in $\mathcal{G}$.

Another approach that has been used to generate system behaviour is described in 4.3. Section 5.3 includes the evaluation for this tool.

4.2 BeSpaceD Random Generator

We have implemented the code that is responsible to generate random BeSpaceD constructs using techniques from functional programming. The Invariant generator is composed of smaller generators such as integer and string generators as shown in the code below:

```
trait Generator[+T] {
  self =>

  def generate: T

  def map[U](f: T => S): Generator[U] = new Generator[U] {
   def generate = f(self.generate)
  }
}

val integers = new Generator[Int] {
   def generate = scala.util.Random.nextInt()
}

val booleans = integers.map(_ >= 0)
val strings = integers.map(_.toString)

def bSpaceD: Generator[Invariant] = for  {
    int1  <- integers
    int2  <- integers
    int3  <- integers
    int4  <- integers
    int5  <- integers
    str    <- strings
} yield IMPLIES(AND(TimeInterval(int1, int2),Owner(str)),
```

```
      OccupyBox(int3, int4, int5, int6))
```

One advantage for developing these tools using pure functional style is the connivance of equational reasoning. The following references include more information about the importance of equational reasoning in developing programs; reasoning about partial and infinite structures (Danielsson and Jansson [2004]), relational and point-free approaches (Bird and De Moor [1996]) and reasoning about effects (Hoareetal [2001]).

4.3 System Behaviour Generator

As a first step to connect the API under test to the behaviour generated by formulas, we have to generate the behaviours themselves. For example, as mentioned before, TLA+ *init* and *next* formulas can be captured by following BNF grammar:

```
TLAFormula := TLAInit | TLANext
TLAInit :=  TLAVar == TLAValue
TLANext := TLAVar == TLAValue | TLAPrimed == TLAValue
```

Using Haskell programming language, it is straight forward to translate the BNF grammar to working code. Therefore,we use Haskell as a first step to gradually implement the required behaviour generator in Scala.

We start with simple example, that of *one bit clock*. The goal of this example is to show how we would generate behaviours from FM formulas. Firstly we present the formulas as data structures:

```
type Primed = Int
type UnPrimed = Int
type PossibleValues = [Int]

data Var = A | B | C
   deriving (Show,Eq)
data Formula = Init [(Var, PossibleValues)] | Next [(Var, UnPrimed, Primed)]
data State  = State [(Var, Int)]
  deriving (Show)

type Behaviour = [State]

initFormula = Init [(B,[0, 1])]
nextFormula = Next [(B, 0, 1), (B, 1, 0)]
```

Then we implement functions that takes formulas and produce Behaviours according to these formulas:

```
initStates :: Formula -> Behaviour
initStates (Init xs) = [State [(var, val)] | (var, vals) <- xs,
                                                val <- vals]

behavForOneInit :: State -> Formula -> Behaviour
behavForOneInit s@(State [(ivar,i)]) (Next ((nvar,up,p):_)) =
 State [(nvar,p)] : behavForOneInit s (Next [(nvar,p,up)])

next :: State -> Formula -> State
next (State xs) (Next ys) = State [(v, prime v val) | (v,val) <- xs]
            where prime v' val = head [p | (vn,up,p) <- ys, v'==vn, val == up]
```

At this stage, we have implemented a behaviour generator for a *one bit clock*. Similar pattern is taken when trying to translate from FOCUSSTspecification to the host programming language. For *One Bit Clock* example, it is enough to show how the model pieces fit together. In other settings, and when specifying more complex systems, the behaviour generator can be extended to accommodate for modeling these systems.

Another approach that can be used to generate the state of behaviour is to re-implement the TLA model checker inside of the programming language as described in section 4.1.

4.4 API Permutation of System Under Test

System‘s API has to be permuted to carry out the testing phase of the framework. These permuted calls can be used to check whether they are actually subsets of behaviours generated by formulas as explained in section 4.3. Therac25 is used as an example to explain the purpose in this regard.

To start generating these permutations, we declare system‘s API as algebraic data type:

```
data Operation = CursorUp | SelectPhotonMode | SelectElectronMode | OtherOperation
```

Then a number of combinatorial functions are defined. These functions are responsible of returning all possible lists that satisfy certain properties. The function *subsequences* return all subsequences of a list, which are given by all possible combinations of excluding or including each element of the list. The functions interleave returns all possible ways to inserting a new element into a list. The function permutations, returns all permutations of a list which can be achieved by the all reorderings of the elements:

```
subsequences [] = [[]]
subsequences(x:xs) = yss ++ map (x:) yss
    where yss = subsequences xs

interleave x [] = [[x]]
interleave x (y:ys) = (x:y:ys) : map(y:) (interleave x ys)

permutations [] = [[]]
permutations (x:xs) = concat (map (interleave x) (permutations xs))
```

For instance, *subsequences [CursorUp, SelectPhotonMode, SelectElectronMode]* gives the following list:

```
[[], [SelectElectronMode], [SelectPhotonMode], [SelectPhotonMode, SelectElectronMode],
 [CursorUp], [CursorUp, SelectElectronMode], [CursorUp, SelectPhotonMode],
 [CursorUp, SelectPhotonMode, SelectElectronMode]]
```

Similarly, *interleave CursorUp [SelectPhotonMode, SelectElectronMode, OtherOperation]* gives the following result:

```
 [[CursorUp, SelectPhotonMode, SelectElectronMode, OtherOperation],
  [SelectPhotonMode, CursorUp , SelectElectronMode, OtherOperation],
  [SelectElectronMode, SelectPhotonMode,CursorUp, OtherOperation],
  [SelectElectronMode, SelectPhotonMode, , OtherOperation,CursorUp]]
```

On the other hand, *permutations [CursorUp, SelectPhotonMode, SelectElectronMode]* results in the following list:

```
[[CursorUp, SelectPhotonMode, SelectElectronMode],
 [SelectPhotonMode, CursorUp, SelectElectronMode],
 [SelectPhotonMode, SelectElectronMode, CursorUp],
 [CursorUp, SelectElectronMode,SelectPhotonMode],
 [SelectElectronMode, CursorUp, SelectPhotonMode],
 [SelectElectronMode, SelectPhotonMode, CursorUp]]
```

Finally, we need a function that returns all of system's API combinations, this is provided by *combinations* function. The definition of which is as follows:

```
combinations = concat . map permutations . subsequences
```

For example, *combinations [CursorUp, SelectPhotonMode, SelectElectronMode]* results in the following API combinations:

```
[[], [SelectElectronMode], [SelectPhotonMode], [SelectPhotonMode, SelectElectronMode],
[SelectElectronMode, SelectPhotonMode], [CursorUp], [CursorUp, SelectElectronMode],
[SelectElectronMode, CursorUp], [CursorUp, SelectPhotonMode], [SelectPhotonMode, CursorUp],
[CursorUp, SelectPhotonMode, SelectElectronMode],[SelectPhotonMode, CursorUp, SelectElectronMode],
[SelectPhotonMode, SelectElectronMode, CursorUp],[CursorUp, SelectElectronMode, SelectPhotonMode],
[SelectElectronMode, CursorUp, SelectPhotonMode],[SelectElectronMode, SelectPhotonMode,CursorUp]]
```

It is this list that the framework uses to exercise all possible API combinations to test against FM formula that has been used to generate the expected behaviours. Evaluation of this tool is included with the *Steam Boiler* evaluation in Section 5.1

4.5 ScalaCheck Extension

ScalaCheck is a PBT library. In ScalaCheck, testing can be applied to more than ordinary properties such as immutable functions. For ordinary properties, ScalaCheck generates random input values and evaluates a boolean expression. For stageful systems, the input can be a sequence of commands and each command‘s post-condition gets evaluated. However, for the purpose of this thesis, an extension to the library has been developed to allows the use of behaviour generated by FM specifications (states) instead of ScalaCheck default state management used in testing state-ful systems such databases and distributed networks.

4.6 Usability

In order to increase the adoption rate of this framework, engineers can be exposed to the ideas presented in this framework gradually. In this section we present a technique to help familiarise the engineers with such ideas. We use the Scytale encryption method as an example.

Programming languages can make the task of programming more difficult than necessary because they have been designed without careful attention to human-computer interaction issues (Newell and Card [1985])

Many introductory algorithm books teach the subject using this style. This style is not only difficult but actually is unnatural to the way human think (Green and Petre [1996]). For example, summing a list in java or C require lots of syntactic distractions that are not relevant to the algorithm. For instance, summing a list of integers in C or java require the use of variable assignments. On the other hand, higher languages such Haskell can solve the same problem in one line using the sum function.

The reason of this assignment is the influence of the Von Neumann machine architecture had in writing software. However, Backus [1978] in his seminal paper called for the liberation of this style through the use of higher level abstractions. He proposed the use of ideas from functional programming paradigm.

FMs incur the use of an even higher level of abstraction. The technique actually embrace state mutation. The mutation is governed by high level simple formulas that engineers are going to be taught beforehand on how to utilise. These formulas is going to be presented through numbered steps without going into details about the underlying mathematics. This approach well help engineers embrace the method without knowing the underlying theory and as a result will indirectly improve the adoption rate.

For example, the following imperative code that encrypt any text using *scytale* encryption method in javascript:

```
function scytale(s){
removeSpaces(s);

var key = 5;
var num = Math.ceil(input1.length/key);
var remaining =input1.length%key;
var result

for(var k=0;k<num;k++){
    for(var j = 0 ; j < key ; j ++){
        var index = j*num+k;
        if(j > remaining && remaining!=0){
            index=index - (j-remaining);
        }
        if(k*key+j < input1.length){
            var b = input1.charAt(index);
            result += b
        }
    }
  }
  return result
}
```

This code resembles the vast majority of the code written in any imperative language. On the other hand,The following code shows the result after designing the same algorithm using the proposed technique:

```
function scytale(s) {
  removeSpaces(s);
  var state = ""
```

```
  var advancer = 0;
  var key = 5;

  /* init */
  state += s.charAt(advancer);
  var num =  Math.ceil(s.length/key);

  /* next */
  while (state.length != s.length) {
      if (advancer + num < s.length) {
        advancer = advancer + num;
      } else  if (advancer + num >= s.length) {
        advancer = 1 + (advancer + (key - 1)) % s.length;
      }
      state += s.charAt(advancer);
  }
    return state;
}
```

This example shows how the nested for-loops where eliminated leading to a code that is declarative and intension revealing. In addition, the use of this method helps in designing the algorithm in quick manner due to the fact that engineers will focus on the *What* (changes the state) instead of the *How*.

It is worth noting that although we use the term *state*, our use of the term should not be conflated with term used in the Von Neumann languages. Our use of states comes from the The Standard Model which asserts that a state is an assignment of values to variables. In the standard model, these states are used to describe behaviours of systems. In our proposed method, the system is the algorithm.

The steps to take in this technique are:

- draw a set of behaviour
- Introduce variables for the things that changed from state to another
- draw simple steps that govern the change
- generalise the step.

Chapter 5

Evaluation

In this section we show the evaluation of the proposed framework. We use two real world examples in Section 5.1 and 5.2 in which we show the workflow and PBT results of the proposed framework. Section 5.3 is dedicated for general performance evaluations of some of the developed tools to measure their scalability.

Firstly, we start with an example of Steam Boiler (Broy and Stølen [2001a]). This example has been chosen because it is a known example in FM literature and exposes most of the functionalities of the proposed framework. For this example, we start by given the TLA+ and FOCUSSTspecification which gets translated to Scala programming language before feeding the translated specification to the proposed framework. The translation correctness is verified manualy by checking the behaviour that is generated by the tool developed in Section (4.1) with the behaviour generated by the actual TLA model checker (TLC). Secondly, we use a simulation of robotic arm that mimics some of the robotic arms that is installed in the Virtual Experiences Lab(VXLab) at RMIT University, Australia. Finally, we provide performance statistics for the behaviour generator and PBT permutations.

5.1 Steam Boiler Simulation

Figure 5.1 and Figure 5.2 show FOCUSSTspecification for a steam boiler. The steam boiler has a water tank, which contains a number of gallons of water. It also contain a pump which adds 10 gallons of water per time unit to its water tank only if the pump is on. On the other hand, if the pump is off, at most 10 gallons of water are consumed per time unit by the steam production. The steam boiler has a sensor that measures the water level. Initially, the water level is 500 gallons, and the pump is off.

The specification group SteamBoiler consists of the following components: SystemReq which includes general requirements specification, ControlSystemArch that contains system architecture, SteamBoiler, Converter, and Controller. The state of steam boiler pump is defined with following type:

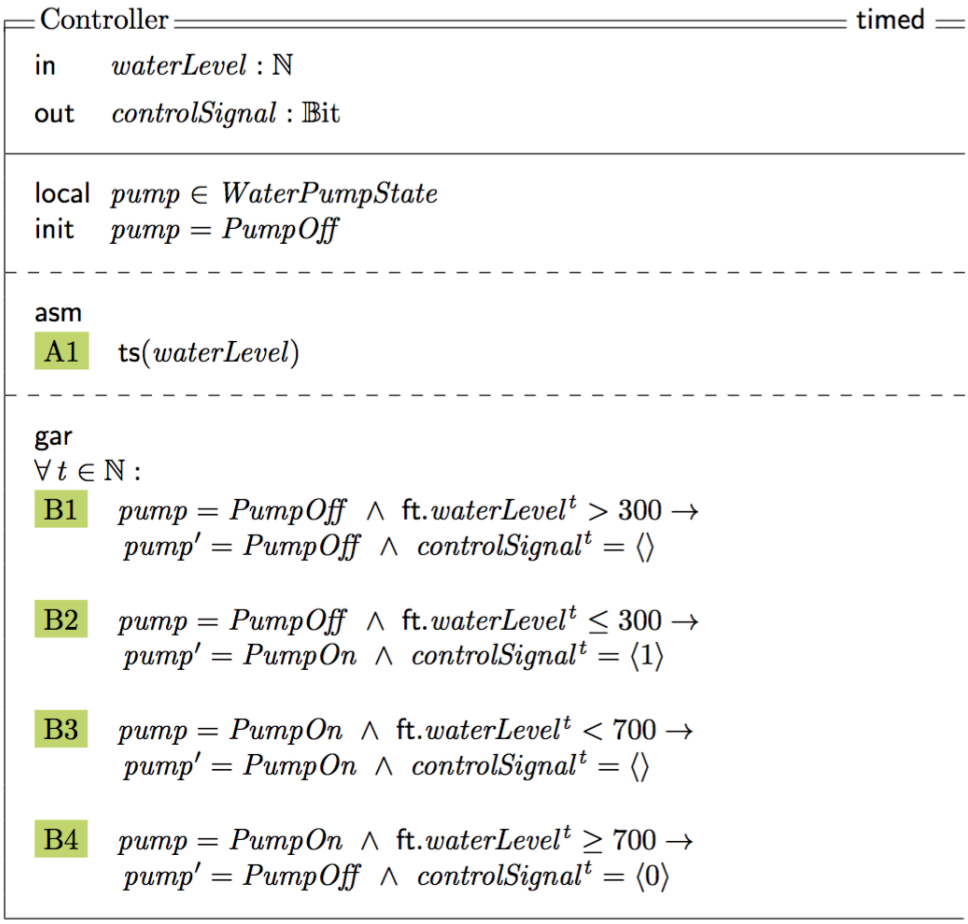


Figure 5.1: FOCUSST *Specification of Steam Boiler Controller (Spichkova [2016])*

```
type WaterPumpState = PumpOn | PumpOff
```

The specification *Controller* as shown in Figure 5.1 describes the controller component of the system. The controller role is to switch the steam boiler pump on and off. In addition, it knows the current state of the pump. The behaviour of this component is asynchronous to keep the number of control signals as small as possible.

Figure 5.2 shows the specification *SteamBoiler* which describes steam boiler component. It has to control the current water level every time interval. The initial water level is specified to be 500 gallons. For every point of time the following must be true:

- if the pump is off, the boiler consumes at most 10 gallons of water
- if the pump is on, at most 10 gallons of water will be added to its water tank.

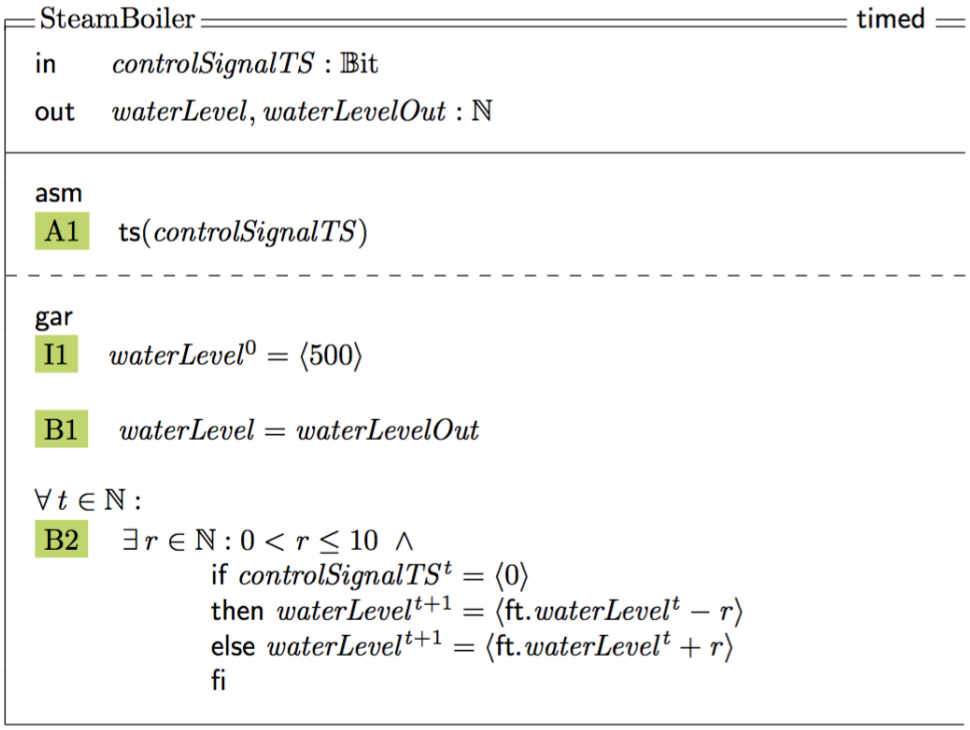


Figure 5.2: FOCUSST *Specification of Steam Boiler group (Spichkova [2016])*

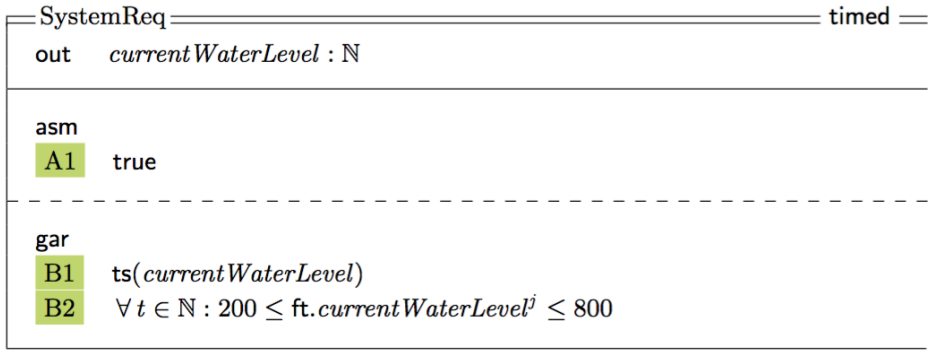


Figure 5.3: FOCUSST *Boiler system requirement (Spichkova [2016])*

The specification SystemReq as shown in Figure (5.3) describes the requirements for the steam boiler system. For each time interval, the system outputs its current water level in gallons and this level should always be between 200 and 800 gallons and the system outputs the information on the water level.

The *Converter* component simply converts the asynchronous output produced by the controller to time-synchronous input for the steam boiler.

Figure 5.4 shows TLA+ specification of the Steam Boiler controller. Unlike FOCUSST, TLA+ is weakly typed. Therefore, it uses a convention to indicated types of variables using *TypeOK* keyword as shown in the specification.

```
MODULE SteamBoiler
LOCAL INSTANCE Naturals
LOCAL INSTANCE Sequences
LOCAL INSTANCE Reals

OneOf(s) ≜ {⟨s[i]⟩ : i ∈ DOMAIN s}
tok(s)   ≜ {⟨s⟩}
Tok(S)   ≜ {⟨s⟩ : s ∈ S}

CONSTANT PumpOn, PumpOff
VARIABLE pump, waterLevel, controlSignal

TypeOK ≜ pump ∈ {PumpOn, PumpOff}
         ∧ waterLevel ∈ Nat

Init ≜ pump = PumpOff
Next ≜ ∨ ∧ pump  = PumpOff ∧ waterLevel > 300
         ∧ pump' = PumpOff ∧ controlSignal = ⟨⟩

       ∨ ∧ pump  = PumpOff ∧ waterLevel ≤ 300
         ∧ pump' = PumpOn ∧ controlSignal = ⟨1⟩

       ∨ ∧ pump  = PumpOn ∧ waterLevel < 700
         ∧ pump' = PumpOn ∧ controlSignal = ⟨⟩

       ∨ ∧ pump  = PumpOn ∧ waterLevel ≥ 700
         ∧ pump' = PumpOff ∧ controlSignal = ⟨0⟩
```

Figure 5.4: TLA+ specification of the Steam Boiler controller

Two real system implementation for the Steam Boiler has been used, correct and wrong implementations. The Correct implementation with behaviour corresponding to the FOCUSSTand TLA+ specifications. The wrong implementation does not correspond to the specifications. For instance, in the case when the system is specified to have its current water level be between 200 and 800 gallons, the wrong implementation does not satisfy this spec and instead have the the current level below 200 and above 800. The wrong example also include the failure of the pump to turn on or off. The Scala APIs for the simulations that has been used include; *startSystem(), endSystem(), pumpDidOpen(), openPump(), pumpDidClose(), closePump(), waterLevelDidChange(amount: Int), checkWaterLevel()* and *controlSignalDidChange(val: Int)*

Table 5.1 shows number of invocations for every API call in each test run. Both translated TLA+ and FOCUSSTspecifications has similar numbers since the schematic translation from TLA+ and FO-

CUSST to Scala is similar in both cases. The extended ScalaCheck implementation that we developed does not shrink the test case to generate minimal failing test cases (which would make the code easier to debug). However, Future work will include the shrinking behaviour that is inspired by *QuickCheck* library.

Table 5.1: Number of API Invocations In Test Cases

API Code	TLA+	FOCUSST
startSystem()	1	1
endSystem()	1	1
pumpDidOpen()	27	27
openPump()	11	11
pumpDidClose()	17	17
closePump()	47	47
waterLevelDidChange(amount: Int)	21	21
checkWaterLevel()	20	20
controlSignalDidChange(val: Int)	26	26

Table 5.2 contrast the performance of permutations and BPT test runs between the schematic translation of TLA+ and FOCUSST. There are no observable differences between the performance of TLA+ and FOCUSST in almost all of the phases of the workflow. This is expected since both TLA+ and FOCUSST has similar syntax and the translation is similar in most cases.

Table 5.2: Translated TLA+ and FOCUSST Statistics

	TLA+	FOCUSST
API permutations in Seconds	10-11	10-11
Behaviour Generator time in Seconds	7-8	7-8
Single Test run time in seconds	0.5	0.5
Total Test run time in seconds for 100 test cases	23-25	23-25

For the same reason (i.e., the translation was similar in both cases), Table 5.3 shows no huge difference between lines of code after translation from TLA+ to Scala and from FOCUSST to Scala.

Table 5.3: Lines of Scala Code After Translation

	Lines of Scala Code
TLA+	70
FOCUSST	75

5.2 Robotic Arm Simulation

This evaluation is initially inspired by case studies that involves robotics that are installed in the VXLab. The initial setup was to plan an installation of API code for test on actual robotic arms. The setup required the help different expertise to be involved in the study. Hardware engineers, VXLab assistant, robotic arms software engineers. Therefore, a simulation of the setup was used instead. For future work, the implemented model will be installed on actual robotic arms in VXLab. For this example, only TLA+ was used. However, FOCUSST could have been used instead.

In the simulation, the function *initialisePosition(): Future[Position]* is responsible to move the robotic arm to an initial position. The *Future* data type is used because moving arms takes long time and we need to verify the final position the arm reached after the API call. However, since *initialisePosition()* is just returning the initial position, it will return instantly. The framework calls this API function and simultaneously check whether it is in accordance to the specified state. Failing tests for the framework might indicate:

- Failure in the software of the system under test. This is one of the benefits of PBT. The found error may have never been discovered otherwise.
- Wrong specification. The system under test may have been wrongly underspecified. In this case, the engineer might change the formulas to reflect system required properties.

The input to the framework in this example is TLA+ formulas and the output is the correct behaviours specified by these formulas. The formulas are written in host programming language (Scala). For example, the initial state for the aforementioned robotic example is specified as follows:

```
val position: TLAVariable = TLAVariable(Y)
val init: TLAInit =  position
```

For this example, the only possible correct behaviour for this specification formula is that *position* should equal to "Y". The framework checks whether the position was indeed "Y" after the call to *initialisePosition()*, otherwise, it reports an error.

Table 5.4 shows some examples for the evaluation of the framework using TLA+ formula. The first call to *initialisePosition()* is correctly specified and the actual result reflects the specification (assum-

Table 5.4: Evaluating cases with TLA+ Init Formulas

API Code	Init Formula	Result	Error?
initialisePosition()	TLAVariable(Y)	Y	No
initialisePosition()	TLAVariable(Y)	K	Yes
moveToQ()	TLAVariable(Q)	Q	Yes
moveToR()	TLAVariable(Q)	M	Yes

ing arm initial position is "Y"), as a result, it is regarded as a successful case. The second call to *initialisePosition()* is different from the actual position, therefore, its was reported as an error. Although the result is expected for the call to *moveToQ()* in the third case, the framework reports an error because the specification is not correct (the arm can not logically move to its current position). Finally, *moveToR()* is reported as error because the actual result (reached position) is not correct. The result column is calculated by getting the value from the *Future* dataype that each API call returns through *onComplete* callback as follows:

```
initialisePosition() onComplete {
  case Success(position) => println(position)
  case Failure(t) => println("An error has occured: " + t.getMessage)
}
```

5.3 Behaviour Generator and PBT statistics

For this section we use simple examples to evaluate the performance of some of developed tools. Most of these examples are used in TLA+ literature to explain its concepts. The examples are *One Bit Clock, DieHard, Euclid algorithm* to calculate *Great Common Divisor* (GCD) and *Therac25*.

One Bit Clock simply alternates between 0 and 1. Such a clock is used to control any modern computer. Its times being displayed as the voltage on a wire. Therefore, there are only two states; the *0 state* and the *1 state*. The *DieHard problem* from the movie*Die Hard 3*, the heroes had to solve the problem of obtaining exactly 4 gallons of water using a 5 gallon jug, a 3 gallon jug, and a water faucet. Euclid's algorithm is an algorithm for computing the greatest common divisor of two positive integers. However, the version used here is simpler than the actual algorithm. It can be described informally as follows: Let the positive integers M and N whose GCD need to be computed.The algorithm uses two variables x and y:

- Start with x equal to M andy equal to N
- Keep subtracting the smaller of x and y from the larger one, until x and y are equal.
- When x and y are equal, they equal the gcd of M and N

Finally, Therac25 example has been explained in the introduction and the background sections

Table 5.5 shows information about the graph that is constructed during the procedure for computing all possible behaviours described in 4.1. *Diameter* column is the number of states in the longest path of the graph in which no state appears twice. *States Found* column is the total number of states it examined in the first step of the algorithm or as successor states in the second step. *Distinct States* column is the number of states that form the set of nodes of the graph. For instance, in case of *One Bit Clock*, model checker found two distinct states. The diameter of one means that largest number of steps is one (transitions from one state to the next) that an execution of the one-bit clock can take before it repeats a state.

Table 5.5: Behaviour Generator Statistics

Example	Diameter	State Found	Distinct States
DieHard	9	97	16
One Bit Clock	1	4	2
Euclid Algorithm	3	22	8
Therac25	9	97	16

The running time of the test cases is shown in Table 5.6. The number shown in this table represent a whole run which involves more than 100 tests cases depending on the number of operations used to model the real system‘s API. Single test case takes less than one seconds to run in almost all examples.

Table 5.6: Running Tests statistics

Example number	Number of API Operations	Running Time(seconds)
DieHard	7	38.085
One Bit Clock	5	12.808
Therac25	8	122.763

All test examples for all the implementations was carried out on two machines, the first of which has the following specifications:

```
Processor Name: Intel Core i5
```

```
Processor Speed: 2.6 GHz
Number of Processors: 1
Total Number of Cores: 2
L2 Cache (per Core): 256 KB
L3 Cache: 3 MB
Memory: 8 GB
Boot ROM Version: MBP111.0138.B14
SMC Version (system): 2.16f65
```

The other machine that was used as well to run some of test examples has the following computing configuration:

```
Processor: Intel Core-i7 360QM 2.0 GHz
Graphics: Nvidia GTX 450M 1 GB
RAM: 4GB DDR3-1333
Hard Drive: 750GB 7,200RPM
Optical Drive: CD-R/RW/DVD-R/RW/+R/+RW+R
Operating System: Windows Premium 64-bit
Connections: 3x USB 2.0, 1x USB 3.0, HDMI, SD Card Reader, Headphone Jack, Mic Jac
Internet connection:  Gigabit Eithernet.
Mouse: Genuine Dell 9RRC7 Black Optical USB Wired 3-Button Plug & Play Mouse With Scroll Wheel,
Other software: CamStudio - Screen Recorder

Display Devices
Screen size: 15.6?
Resolution: 1366x 768
Frequency:  60Hz
Colour: 16 million
```

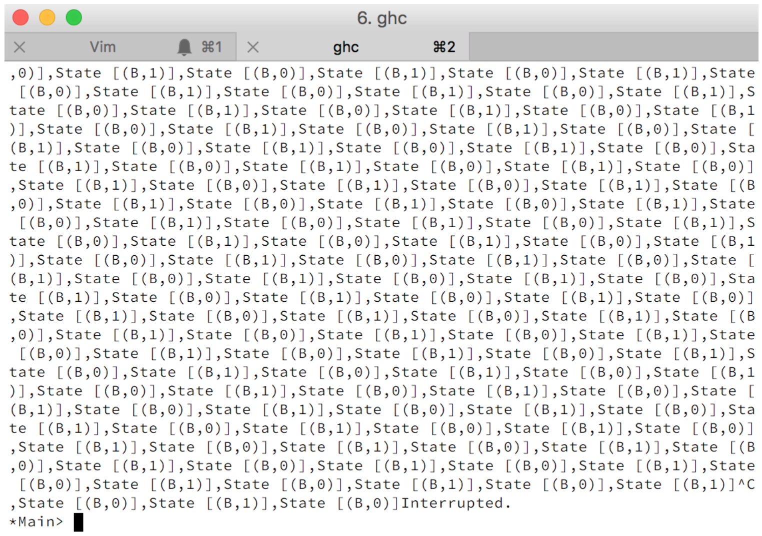


Figure 5.5: Behaviour generated from formulas

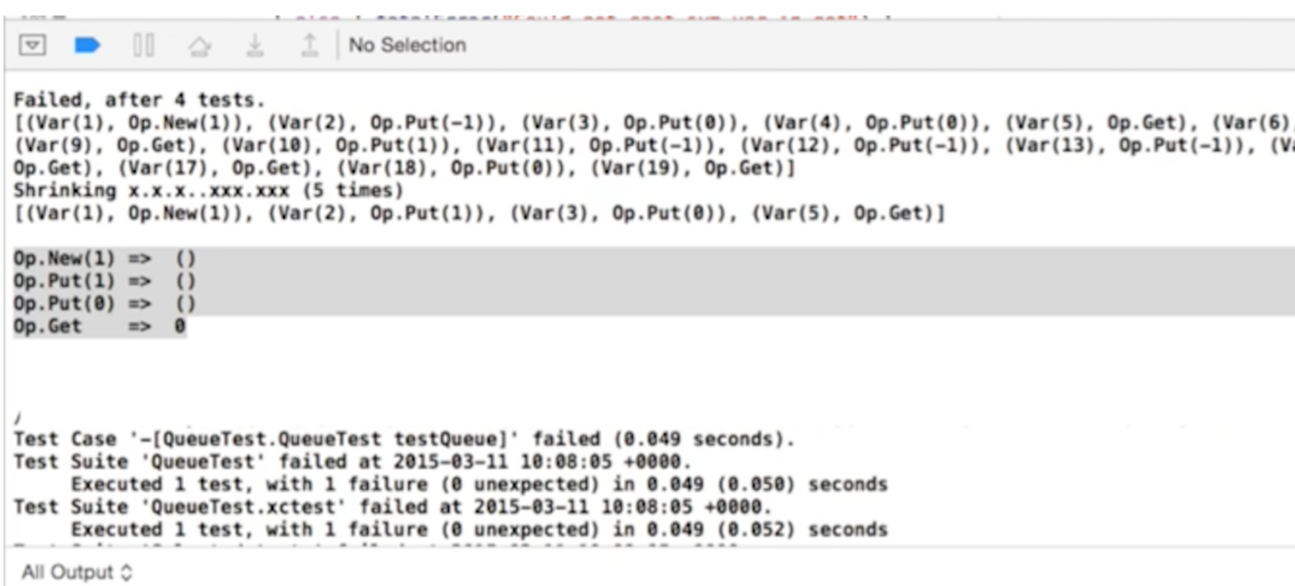


Figure 5.6: Failing Tests

Figure 5.7: Passing Tests after changing system code

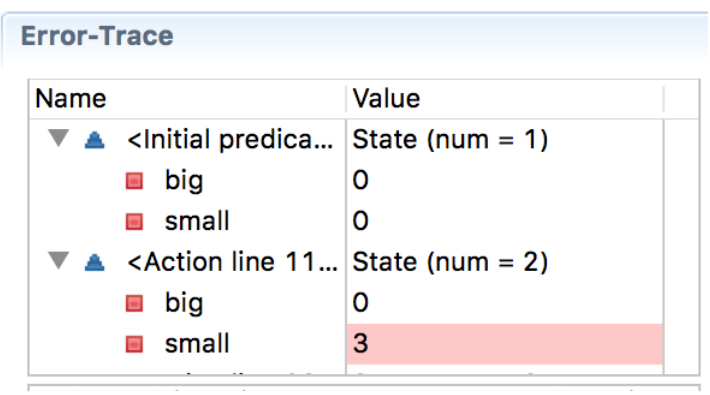


Figure 5.8: TLA Model Checker error

Chapter 6

Conclusions and Future Works

6.1 Conclusions

Many software engineers abstain from the use of formal methods when designing systems despite its great advantages. However, they are usually comfortable with concepts from property-based testing that require a little bit of mathematical thinking. property-based testing allows for the use of randomly generated test cases based on properties to test systems against their specifications. John Hughes –the co-inventor of QuickCheck– has shown that using this library has allowed him to discover hundreds of bugs in critical systems such as automobiles and DropBox file sharing service. However, QuickCheck uses Haskell programming language specific constructs (such as arrays, integers) and more complicated data types (such as classes and algebraic data types) to model the specification of a system. In this thesis, we have presented our work on the use of temporal models for formal methods-based analysis and property based testing instead of these simple constructs. We have described different ingredients and their interplay: testing frameworks, TLA+, FOCUSST and BeSpaceD. We have developed tools as stepping stones. The overall goal of our research is the reduction of the impedance mismatch between formal methods and practitioners through the integrations of formal methods with property-based testing.

6.2 Future Works

Extending the framework to work with other asynchronous systems such as *user interfaces* and other *event based* systems. Asynchronous programming using the Observable data type can make compositions of such events easier (Meijer [2010]).

Refactoring *Future* data type to use plain *Monad*. This will make substituting *Monad* instances easier. For instance, in the context of testing, we can use the *Id* monad, and in the context of production code, we can use the *Future* Monad.

Improving the performance of the framework implementation using ideas from *Programs for cheap!* (Hackett and Hutton [2015]).

The manual translation of FM specification to Scala can be made automatic in future work. In addition, future work can include extending the framework to allow for representing FM formulas in strings of characters instead of the constructs in the host language.

The human-oriented usability technique presented in section 4.6, can be extended to study its effectiveness in teaching algorithms to novice programmers.

The random generation of test cases is not being shrinked to show minimal failing test cases. The shrinking behaviour is being used in QuickCheck and ScalaCheck implementations. Therefore, future work should include shrinking in the extended version of ScalaCheck to make the code easier to debug.

Extending the categorical formalism to explore possible connections with other branches of mathematics such as *Topology* and *Homotopy Type Theory* (Awodey [2015])

Bibliography

N. Alzahrani, M. Spichkova, and J. Blech. Spatio-temporal model for property based testing. In *3rd International Workshop on Human-Oriented Formal Methods (HOFM 2016)*, LNCS. Springer, 2016.

A. Arcuri and L. Briand. Adaptive random testing: An illusion of effectiveness? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 265–275. ACM, 2011.

S. Awodey. Homotopy type theory. In *Indian Conference on Logic and Its Applications*, pages 1–10. Springer, 2015.

J. Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

R. Bird and O. De Moor. The algebra of programming. In *NATO ASI DPD*, pages 167–203, 1996.

J. Blech, I. Peake, H. Schmidt, M. Kande, A. Rahman, S. Ramaswamy, S. Sudarsan, and V. Narayanan. Efficient Incident Handling in Industrial Automation through Collaborative Engineering. In *IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*. IEEE Computer, Sept 2015.

J. O. Blech. An example for BeSpaceD and its use for decision support in industrial automation. 2015. URL `http://arxiv.org/abs/1512.04656`.

J. O. Blech and H. Schmidt. BeSpaceD: Towards a tool framework and methodology for the specification and verification of spatial behavior of distributed software component systems. 2014. URL `http://arxiv.org/abs/1404.3537`.

J. P. Bowen and M. G. Hinchey. Seven more myths of formal methods. *IEEE software*, 12(4):34, 1995.

M. Broy and K. Stølen. Specification and development of interactive systems: focus on streams, interfaces, and refinement. 2001a.

M. Broy and K. Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer, 2001b.

R. N. Charette. Why software fails [software failure]. *Spectrum, IEEE*, 42(9):42–49, 2005.

T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. Tse. Adaptive random testing: The art of test case diversity. *Journal of Systems and Software*, 83(1):60–66, 2010.

K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of haskell programs. 46(4):53–64, 2011-05. ISSN 0362-1340. doi: 10.1145/1988042.1988046. URL `http://doi.acm.org/10.1145/1988042.1988046`.

N. A. Danielsson and P. Jansson. Chasing bottoms. In *International Conference on Mathematics of Program Construction*, pages 85–109. Springer, 2004.

B. Dhillon. *Engineering Usability: Fundamentals, Applications, Human Factors, and Human Error*. American Scientific Publishers, 2004.

A. Gerdes, J. Hughes, N. Smallbone, and M. Wang. Linking unit tests and properties. In *Proceedings of the 14th ACM SIGPLAN Workshop on Erlang*, Erlang 2015, pages 19–26. ACM, 2015. ISBN 978-1-4503-3805-9. doi: 10.1145/2804295.2804298. URL `http://doi.acm.org/10.1145/2804295.2804298`.

J. A. Goguen. A categorical manifesto. *Mathematical structures in computer science*, 1(01):49–67, 1991.

T. R. G. Green and M. Petre. Usability analysis of visual programming environments: a cognitive dimensions framework. *Journal of Visual Languages & Computing*, 7(2):131–174, 1996.

J. Hackett and G. Hutton. Programs for cheap! In *Proceedings of the 2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 115–126. IEEE Computer Society, 2015.

M. G. Hinchey. Confessions of a formal methodist. In *Proceedings of the Seventh Australian Workshop Conference on Safety Critical Systems and Software 2002 - Volume 15*, SCS '02, pages 17–20. Australian Computer Society, Inc., 2003.

C. Hoareetal. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. *Engineering theories of software construction*, 180:47, 2001.

S. Hordvik, K. Øseth, J. O. Blech, and P. Herrmann. A Methodology for Model-based Development and Safety Analysis of Transport Systems. In *11th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, 2016.

Z. Hu, J. Hughes, and M. Wang. How functional programming mattered. *National Science Review*, 2 (3):349–370, 2015.

J. Hughes. Software testing with quickcheck. In *Central European Functional Programming School*, pages 183–223. Springer, 2010.

L. Lamport. Hybrid systems in TLA+. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, number 736 in Lecture Notes in Computer Science, pages 77–102. Springer Berlin Heidelberg, 1993. DOI: 10.1007/3-540-57318-6_25.

L. Lamport. The temporal logic of actions. 16(3):872–923, 1994. ISSN 0164-0925. doi: 10.1145/177492.177726. URL http://doi.acm.org/10.1145/177492.177726.

L. Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

E. Meijer. Reactive extensions (rx): curing your asynchronous programming blues. In *ACM SIGPLAN Commercial Users of Functional Programming*, page 11. ACM, 2010.

E. Miller. The Therac-25 Experience. In *Conf. State Radiation Control Program Directors*, 1987.

C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How amazon web services uses formal methods. *Communications ACM*, 58(4):66–73, Mar. 2015. ISSN 0001-0782. doi: 10.1145/2699417. URL http://doi.acm.org/10.1145/2699417.

A. Newell and S. K. Card. The prospects for psychological science in human-computer interaction. *Human-computer interaction*, 1(3):209–242, 1985.

T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

S. Patra. Worst-case software safety level for braking distance algorithm of a train. In *System Safety, 2007 2nd Institution of Engineering and Technology International Conference on*, pages 206–210. IET, 2007.

F. Redmill and J. Rajan. *Human factors in safety-critical systems*. Butterworth-Heinemann, 1997.

D. E. Rydeheard and R. M. Burstall. *Computational category theory*, volume 152. Prentice Hall Englewood Cliffs, 1988.

M. Spichkova. Human Factors of Formal Methods. In *In IADIS Interfaces and Human Computer Interaction 2012*. IHCI 2012, 2012.

M. Spichkova. *Design of formal languages and interfaces: "Formal" does not mean "unreadable"*. IGI Global, 2013.

M. Spichkova. Human factors of formal methods. *arXiv preprint arXiv:1404.7247*, 2014.

M. Spichkova. Spatio-temporal features of focusst. *arXiv preprint arXiv:1610.07884*, 2016.

M. Spichkova, J. O. Blech, P. Herrmann, and H. W. Schmidt. Modeling spatial aspects of safety-critical systems with focus-st. In *MoDeVVa@ MoDELS*, pages 49–58. Citeseer, 2014.

M. Spichkova, H. Liu, M. Laali, and H. W. Schmidt. Human factors in software reliability engineering. *Workshop on Applications of Human Error Research to Improve Software Engineering (WAHESE2015)*, 2015.

M. Spichkova et al. *Specification and seamless verification of embedded real-time systems: FOCUS on Isabelle*. PhD thesis, Technical University Munich, 2007.

Y. Yu, P. Manolios, and L. Lamport. Model checking tla+ specifications. In *Correct Hardware Design and Verification Methods*, pages 54–66. Springer, 1999.

A. Zamansky, G. Rodriguez-Navas, M. Adams, and M. Spichkova. Formal methods in collaborative projects. In *11th International Conference on Evaluation of Novel Approaches to Software Engineering*. IEEE, 2016.