MODEL-BASED SOFTWARE TESTING FOR DISTRIBUTED SYSTEMS AND PROTOCOLS

Doctoral Dissertation by
Rui Wang

Thesis submitted for
the degree of Philosophiae Doctor (PhD)
in
Computer Science:
Software Engineering, Sensor Networks and Engineering Computing

Department of Computer Science,
Electrical Engineering and Mathematical Sciences

Faculty of Engineering and Science

Western Norway University of Applied Sciences

May 4, 2020

Series of dissertation submitted to
the Faculty of Engineering and Science,
Western Norway University of Applied Sciences.

ISBN: 978-82-93677-20-8

Author: Rui Wang

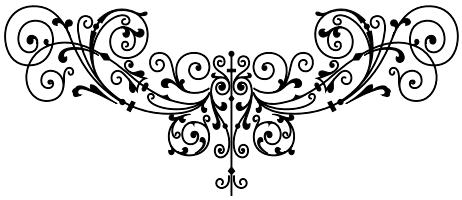
Title: Model-based Software Testing for
Distributed Systems and Protocols

Printed production: The Communication Division /
Western Norway University of Applied Sciences

Bergen, Norway, 2020

TO MY PARENTS,

for endlessly loving, cultivating, supporting, and encouraging me.



PREFACE

The author of this thesis has been employed as a Ph.D. research fellow in the software engineering research group at the Department of Computer Science, Electrical Engineering and Mathematical Science at Western Norway University of Applied Sciences. The author has been enrolled into the PhD programme in Computer Science: Software Engineering, Sensor Networks and Engineering Computing, with a specialization on software engineering.

The research presented in this thesis has been accomplished in cooperation with the Department of Electrical Engineering and Computer Science at the University of Stavanger, Norway, and in cooperation with the School of Electrical Engineering and Computer Science at the KTH Royal Institute of Technology, Stockholm, Sweden.

This thesis is organized in two parts. Part I is an overview article providing an introduction to the research field of model-based software testing for distributed systems and protocols, a discussion of the research methodology used, a summary of the results obtained, and a discussion of the research contributions of this thesis in the context of state-of-the-art related work. Part II consists of four published and peer-reviewed research articles (A-D), and one paper (E) submitted to an international conference:

Paper A R. Wang, L. M. Kristensen, H. Meling, and V. Stolz. Model-Based Testing of the Gorums Framework for Fault-Tolerant Distributed Systems. In Transactions on Petri Nets and Other Models of Concurrency XIII, volume 11090 of Lecture Notes in Computer Science, pages 158–180, Springer International Publishing, 2018.

Paper B R. Wang, L. M. Kristensen, H. Meling, and V. Stolz. Automated test case generation for the Paxos single-decree protocol using a Coloured Petri Net model. In Journal of Logical and Algebraic Methods in Programming, volume 104, pages 254–273, Elsevier Ltd, 2019.

Paper C R. Wang, L. M. Kristensen, and V. Stolz. MBT/CPN: A Tool for Model-Based Software Testing of Distributed Systems Protocols Using Coloured Petri Nets. In Verification and Evaluation of Computer and Communication Systems, volume 11181 of Lecture Notes in Computer Science, pages 97–113, Springer International Publishing, 2018.

Paper D R. Wang, C. Artho, L. M. Kristensen, and V. Stolz. Visualization and Abstractions for Execution Paths in Model-Based Software Testing. In Integrated Formal Methods, volume 11918 of Lecture Notes in Computer Science, pages 474–492, Springer International Publishing, 2019.

Paper E R. Wang, C. Artho, L. M. Kristensen, and V. Stolz. Multi-objective Search for Model-based Testing. Submitted to The 20th IEEE International Conference on Software Quality, Reliability, and Security, Vilnius, Lithuania, IEEE, 2020.

ACKNOWLEDGMENTS

My 4 years of doctoral life in Bergen have been influenced by lots of people in various ways. Here, I would like to give my heartfelt thanks to all. First, I want to give my biggest thanks to my supervisors Lars Michael Kristensen, Volker Stolz, and Hein Meling for their priceless guidance, countless help, infinite patience, and great kindness. I particularly appreciate their diverse guidance for my research work during my doctoral study.

I would like to give a special thank to Lars for helping me to learn and understand modeling with Coloured Petri Nets, for giving reviews and providing suggestions to all my work at every phase of my doctoral study, and pushing and encouraging me when I needed to do better for my research. One benefit for me throughout my Ph.D. study was that my workplace was very close to Lars's office, so it was easy for me to stop by his office to ask questions and have discussions. I was always welcomed by Lars every time. Especially during my work for the thesis, Lars gave invaluable support for the reviews and advice to this thesis numberless times, and we had research meetings almost every week throughout my dissertation. Without the help and support from Lars, I think this thesis would not have been completed on time. Also, I still remember that 4 years ago when I arrived at Bergen airport at 11 PM for this doctoral work, Lars picked me up and helped me to have a good start in Bergen. I really appreciate his kindness for this and since then.

I want to thank Volker for introducing and broadening the research area and work for me. He introduced the knowledge of the runtime verification to me and provided opportunities for me to work with Lego robot, to cooperate with other researchers, and to visit universities and conferences outside Norway. He also very welcomed me to his office and helped me to solve detailed and technical problems during my Ph.D. study. I also want to thank him for helping me to build the connection with KTH so that I could have my research stay there. During the work for this dissertation, Volker also gave plenty of inputs and a huge effort to the format and layout of the thesis and spent time reviewing and having meetings with me. I am also most grateful for his support and help to temporarily hire me for his COEMS project so that I had time to finish our last paper and my thesis. In addition, I was also invited many times to his home for the delicious homemade food. I can never thank him enough for his friendly welcome.

I also want to give my thank to my supervisor Hein in UiS. We have two journal publications collected in this thesis, and Hein contributed in various ways both for the implementation and paper writing. Hein gave me countless and valuable suggestions and pointed out the mistakes for the young researcher like me during our cooperation. He also helped me to further learn the knowledge of distributed systems and consensus protocols, and improved my coding skill in Golang programming language and other related techniques such as Google gRPC and protocol buffers. Because of Hein's help, I got the opportunity to attend a summer school for distributed systems. I was also very welcomed by him when I was visiting UiS for the distributed systems course that he provided. I appreciate his kindness for driving me to the airport after I finished the course.

Not only had I the best supervisors during my doctoral study, but I appreciate the people who helped and supported my research. I would like to give my special thank to Cyrille Artho at KTH. I consider him as my "unofficial co-supervisor" since my midterm evaluation. We have two publications together also collected in this thesis. I want to thank him for hosting me at KTH when I was there as a visiting doctoral researcher. I had a really nice stay for those 5 months at KTH. We had lunch together for almost every working day, also with his colleagues at KTH (thanks to them too!). During my research stay at KTH, Cyrille gave many ideas and suggestions about my research directions and helped me to solve technical problems. We basally had a research meeting every day when I was at KTH. Even after my stay at KTH, we still had very good cooperation for the research and therefore we had two publications, and Cyrille also contributed to these publications in many and diverse ways. Cyrille was also the external examiner for my midterm evaluation. Therefore, I also want to thank him for his time to come to HVL Bergen to attend my evaluation.

In addition to my supervisors, I would like to thank Western Norway University of Applied Sciences (Bergen University College when I was hired) for funding my Ph.D. research fellow position. I am really thankful to the staff at the department of Computer Science, Electrical Engineering and Mathematical Sciences for their friendly support. In particular, I want to first mention Kristin Fanebust Hetland, Håvard Helstrup, and Pål Ellingsen. They always kindly provided me with help and made sure that I was doing well if I met any problems at work. I was also very welcomed by them when I needed to stop by their offices. I am truly thankful for the time and effort they used to support my work in the department. Here, I especially want to thank Håvard together with Lars who both helped and contributed to the Norwegian abstract of the thesis. I would also like to mention Adrian Rutle who was always nice and friendly to me. Thanks to Adrian, for organizing skiing events and providing opportunities for me to try skiing and that was lots of fun. Adrian was also the internal examiner for my midterm evaluation. I thank him for giving lots of valuable feedback for my research. Also, he invited me, Dr. Fernando, and Dr. Ajith to his home for the traditional Norwegian Christmas dinner. That was my first time to try pinnekjøtt. I want to give my another thank to Violet Ka I Pun and Volker together for hosting me for parties and dinners at their lovely home. Thanks also to Yngve Lamo and Talal Rahman, we always had some nice and interesting discussions at work. I would also like to mention Sven-Olai and Harald whom I worked with in the courses I was involved in teaching. It was a very nice experience for me to work with them. Especially, I would like to thank Sven-Olai for helping me to practice Norwegian and to book the bus to Førde for me to give a lecture there.

I am deeply grateful to have Dr. Ajith and Dr. Fernando as my two closet colleagues, also flatmates, and best bros. Ajith is a problem solver and can always provide help and information to me about different things in Norway. Fernando is also a good bro who helped me a lot with many things. I still remember that he showed me around Bergen to introduce different things and places after I just started my job. That day was raining of course. We have also traveled to many places together within and outside Norway. They are my best travel buddies. We are like the Three Musketeers. Thank you for being in my life.

I would also like to mention my Norwegian colleague and friend Simon who was

always willing to have a chat about different things. He also invited me many times to his home with great food provided, especially Nachos. I also give my special thank to Simon's wife Jillian who is also very nice every time we meet. Also, to Lucas, thanks for always being willing to discuss work and life and to give useful opinions. It is worth mentioning that it was very nice of you guys to invite me to different activities. We had fun many movie nights in the cinema, quizzes in the bar, and even curling. I think these were all quite helpful for us to relax from our work stress and pressure.

I also want to give my thanks to all other former and new Ph.D. researchers in our Ph.D. office at HVL, for both the great working and social environment. To Espen and Andreas, thanks for helping me a lot about different information in Norwegian at the beginning of my work at HiB (HVL). To Rabbi, thanks for always being willing to give help and suggestions about research work. To Maxim, it was always fun to chat with you about Norwegian culture, language, and all other interesting things. To Angela and Alex, I appreciate the time we spent together as colleagues and friends. That was so much fun for mini-golf, curling, and movies we went to before. To Patrick, it was always great to have some beers with you and Fernando. To Jarle, although it was a very short time, it was nice to sit next to you and discuss work sometimes. To Justus, thank you for inviting me to your home for dinner; that was a fun night with great discussions. Also, to Suresh, Anton, Mahmood, Håkon, Faustin, Remco, and all other Ph.D. researchers, thanks to you, there were countless funny and interesting topics we have discussed during lunch and coffee breaks or at HVL events and parties.

In addition, I would like to thank my friends in Norway and Sweden for their kindness, support, and help in my personal life. First, in Bergen, I am so thankful to have Iril and Ludmila as friends who provided knowledge of Norwegian culture and traditions and gave useful information about life in Norway to me. I enjoyed the time with you. No matter in the sunshine with the ice cream or a coffee house, every time we can have nice and deep discussions. Thank you also to Iril for inviting me to watch movies in the cinema with her friend Astrid who is also super nice. We all enjoyed the movies and had nice discussions. I want to give my thank to Linnea who invited me to the Christmas party. That was a great night. We made pepperkaker and glögg. I also received a present from her. Also, I want to mention Alejandro (amigo). It was nice to hang out and chill in the sunshine in Nordnes with amigos Alejandro and Ajith. In Sweden, I would like to thank my bro William in Stockholm. When I was doing my research stay at KTH, it was really helpful to have a local Swedish bro like him to introduce all the things in Stockholm, invite me to his home for dinner and barbecue, go through different Viking bars in Gamla Stan.

The year 2019 was a very dark and tough time for me. In January 2019, my mother was diagnosed with breast cancer stage 4 and it has spread to lungs, liver, and bone. My mother also lost the ability to walk because of cancer in bone. Therefore, as the only child in my family, I had to travel back home to China in order to handle this urgent treatment to my mother's sickness. Here, I am truly grateful to those special people who provided kind support, warm help, and great comfort to me for this situation I was in since then. I would like to express my gratitude again to my supervisors Lars and Volker who supported my decision to travel back to China for my mother. At that time I was involved in a course taught by Volker and Sven-Olai, so I want to thank them and Pål together for quickly finding my replacement for the course. Thanks to

Håvard for comforting me after I got the bad news about my mother from China. I am most thankful to Kristin who helped me for taking leaves and always cared about me and my mother's situation. I cannot thank all of you enough for all that you have done for me. The treatment for my mother is not easy all the time. I also needed to help her to do the chemotherapy and targeted therapy in the hospital and to deal with different side effects at home when I was in China. The good news is that the treatment worked and hence giving me some relief, and correctly, My mother's sickness is still under control by the treatment. I hope everything is going to be better for her. I also want to thank Fernando, Ajith, Iril, Ludmila, Astrid, Angela, and Alex who kindly asked and cared about me and my mother's situation. Thank you again for your understanding and caring heart.

At last, I would like to express my heartfelt and eternal gratitude to my parents. I consider you as the most important role model in my life. I will never forget your countless sacrifices to bring me up and your constant care and selfless love to me. I truly appreciate your patiently cultivating and countless times of support and encouragement. You were always there to congratulate me and share my joy when I had good moments and successes; you were also there to listen to my troubles, to comfort me, and to put me back on my feet with valuable advice whenever I had a bad day and desperately need a hand. I will never forget this. Never. I especially want to give deep gratitude to my mother. Thank you for always teaching me to be a person who must have courage and bravery, even during your sickness, and thank you for always being supportive and encouraging me to finish my doctoral study. The love you have shown me is so great. To both of my parents again, I am extremely grateful that you have always stood behind me, guided me in the right direction, and encouraged me to pursue my dreams. Thank you for being the best thing that has ever happened to me. Thanks a billion for everything you have done for me. I wish you could always be happy and healthy.

ABSTRACT

Society is increasingly dependent on fault-tolerant cloud-based services which rely on the correctness and reliability of advanced distributed software systems and consensus protocols. The implementations of these systems require complex processing logic which in turn makes them challenging to implement correctly and also challenging to test in a systematic way. Model-based software testing (MBT) is a powerful approach for testing software systems. MBT enables automated test case generation from models, which can be used to investigate fault-tolerance and expose errors in the implementations of software systems.

The research idea underlying this thesis is to investigate the application of MBT for distributed software systems and protocols, with the aim of ensuring the correctness, reliability, and consistency of their implementations. This thesis contains scientific contributions in three main research areas of MBT for distributed systems.

The first contribution is to investigate MBT for quorum-based fault-tolerant distributed systems. This has resulted in a general MBT approach and supporting QuoMBT framework based on generating test cases from models created via Coloured Petri Nets (CPNs). QuoMBT enables testing of the Gorums middleware framework and quorum-based fault-tolerant distributed systems implemented via Gorums. Our experimental evaluation shows that the QuoMBT framework can obtain high code coverage and successfully detect programming errors in the Gorums middleware and distributed systems implemented based on the Gorums framework.

The second contribution has been to develop software tools and techniques to support MBT for distributed systems. We have developed the MBT/CPN software engineering tool for test case generation from models constructed using CPNs. The tool can perform both simulation and state space-based test case generation, and is important for practical application of MBT. The MBT/CPN tool has been successfully applied to test a distributed storage system and a Paxos consensus protocol both implemented via the Gorums framework. The general applicability of the tool has been demonstrated by validating the correctness of a two-phase commit transaction protocol.

The third contribution involves two research directions. One is an approach to measure and visualize the execution path coverage criterion of test cases. The experimental results show that our abstraction-based visualization provides useful visual feedback of tests, their coverage and diversity. The other is a search-based test case generation technique based on multi-objective reinforcement learning and optimization. It relies on a bandit-based heuristic search strategy implemented to guide test case generation and a multi-objective optimization technique. We have performed an experimental evaluation on a collection of examples, including the ZooKeeper distributed coordination service. The results show that test cases generated using our search-based approach provide predictable and improved state- and transition coverage, find failures earlier, and provide increased path coverage.

SAMMENDRAG

Samfunnet baserer seg i voksende grad på bruk av feiltolerante skytjenester som er avhengig av korrekte og pålitelige avanserte distribuerte programvaresystem og konsensusprotokoller. Implementering av disse systemene krever kompleks prosesseringslogikk som gjør dem utfordrende å implementere korrekt samt utfordrende å teste på en systematisk måte. Modell-basert programvaretesting (MBT) er en kraftfull tilnærming for testing av programvaresystemer. MBT gjør det mulig automatisk å generere testtilfeller fra modeller som kan brukes til å undersøke feiltoleranse og eksponere feil i implementasjon av programvaresystemer.

Forskningsidéen som ligger til grunn for denne avhandlingen er å undersøke bruken av MBT på distribuerte systemer og protokoller med det formål å sikre korrekthet, pålitelighet og konsistens av implementasjon. Avhandlingen inneholder vitenskapelige bidrag i tre hoved-forskningsområder innen MBT for distribuerte systemer.

Det første vitenskapelige bidraget er undersøkelse av MBT for quorum-baserte feiltolerante distribuerte systemer. Dette har resultert i en generell MBT tilnærming og et tilhørende QuoMBT rammeverk basert på generering av testtilfeller fra modeller konstruert ved bruk av Colored Petri Nets (CPNs). QuoMBT muliggjør testing av Gorums-mellomvare og quorum-baserte feiltolerante distribuerte systemer implementert ved bruk av Gorums. Vår eksperimentelle evaluering viser at QuoMBT rammeverket kan oppnå høy kodedekning og oppdage programmeringsfeil i Gorums-mellomvare og distribuerte systemer implementert ved bruk av Gorums-rammeverket.

Det andre vitenskapelige bidraget har vært å utvikle programvareverktøy og teknikker som støtter MBT for distribuerte systemer. Vi har utviklet MBT/CPN programvareutviklingsverktøyet for generering av testtilfeller fra modeller konstruert ved bruk av CPNs. Verktøyet kan generere testtilfeller basert på både simuleringer og tilstandsrom, og er viktig for den praktiske anvendelsen av MBT. MBT/CPN-verktøyet har vært brukt for å teste et distribuert lagringssystem og en Paxos konsensusprotokoll begge implementert via Gorums-rammeverket. Verktøyets generelle anvendbarhet er påvist ved å validere korrektheten av en to-fase commit transaksjonsprotokoll.

Det tredje vitenskapelige bidraget involverer to forskningsretninger. Den ene er en tilnærming for å måle og visualisere dekningskriterium av utførelsessekvenser i programvare for testtilfeller. De eksperimentelle resultatene viser at vår abstraksjonsbaserte visualisering gir nyttig visuell informasjon om programvaretester, deres dekning og mangfold. Den andre er en søke-basert teknikk basert på fler-objektiv forsterkningslæring og optimalisering for å generere testtilfeller. Teknikken bygger på en banditt-basert søkeheuristikk implementert for å styre generering av testtilfeller og en fler-objektiv optimaliseringsteknikk. Vi har utført en eksperimentell evaluering av en samling eksempler, inkludert den distribuerte koordinasjonstjenesten ZooKeeper. Resultatene viser at testtilfeller generert ved bruk av vår søke-baserte tilnærming gir forutsigbar og forbedret tilstand- og transisjonsdekning, finner feil tidligere og gir økt dekning for utførelsessekvenser i programvare.

Contents

Part I

OVERVIEW

We have seen that computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty.

— Donald E. Knuth [73]

CHAPTER 1

INTRODUCTION

Information technology is dramatically altering the way we organize society and conduct business. Today, critical aspects of the most exciting innovations, such as smart energy systems in homes, intelligent transportation systems, health-care systems, and robotic product development systems in industry, increasingly rely on the correctness, availability and performance of cloud-based software services implemented based on advanced distributed systems. These cloud services have a strong influence on not only almost all aspects of our life and on society, but also on the developments in the *Internet of Things* (IoT) domain. The complex logic required in the software of distributed systems and cloud services frequently leads to implementation errors [1]. Such errors can cause the cloud services to become unavailable and even return erroneous results to users.

A key foundation to realize robust distributed systems for cloud services is to apply sophisticated distributed algorithms and consensus protocols. These algorithms and protocols make it possible to handle server failures, to replace failed servers, and to extend the capacity of the system dynamically, without interrupting the services provided to the users. However, distributed algorithms and protocols are notoriously difficult to understand and to implement correctly, and they rely on complex logic and practices that render the development of distributed systems and cloud services error-prone [64, 97, 102, 104]. This means that current software engineering techniques are often inadequate for reliable development of such complex systems and services, and there is an urgent need for better software engineering and development approaches to ensure the correctness of the implementation of these systems and services.

In this thesis, our research idea is to combine model-driven software engineering (MDSE) [21], formal verification, and software testing techniques with a main focus on the application of *model-based software testing (MBT)* [125] for the development of distributed systems and protocols used to implement cloud-based services. We aim to contribute to enabling reliable implementations of distributed systems and protocols based on behavioral models, and to detect and eliminate errors from distributed systems and protocols in order to achieve correctness.

In this chapter, we first discuss the method and techniques we focus on in this thesis to implement fault-tolerant and consistent distributed computing services. Then, we introduce the background of software testing with a specific focus on model-based software testing. We also briefly introduce the modeling formalisms and associated tools that we have used in this thesis. At the end of this chapter, we present and discuss

the research questions underlying this thesis, and introduce the research method that has been applied to address the research questions.

1.1 Fault-tolerant Distributed Computing

In distributed computing, a fundamental problem is to achieve overall system reliability in presence of the failure of some of its processes. *Fault-tolerance* enables a distributed system to continue operating in the presence of a number of faulty processes. This often requires processes to agree on some common data value that is needed during computation in order to ensure the *consistency* of the system. In this section, we discuss the approaches and techniques to enable fault-tolerance and ensure consistency of distributed computing services.

1.1.1 State-machine Replication

In distributed computing, for the sake of ensuring fault-tolerance of services, *state-machine replication* is usually considered as a general approach to implement highly available distributed software services via executing several replicated version of a centralized service (also known as *replicas*) [117]. These replicas are executed on different processors (machines) that are assumed to fail independently, and the client interactions with the replicas are handled by protocols. In this way, the availability of the service is ensured despite the failure of a subset of the processors. Specifically, the approach involves deploying a number of replicated state machines on multiple independent servers; the replicas start in the same initial state and execute the requests from clients in the same order. Each replica on each server therefore performs the same operations and produces the same output result to clients. The key to using the state-machine replication approach is to ensure that all replicas receive and process the same sequence of requests from clients in order to achieve consistency.

1.1.2 Quorum Systems

In distributed computing, given a set of nodes, typically servers, a *quorum system* is a collection of subsets of nodes, called *quorums*, every two of which intersect [126]. That is, quorums come in groups, forming quorum systems. Each quorum can operate on behalf of the system, thus increasing its availability and performance. The intersection property of quorum systems guarantees that operations done on distinct quorums preserve consistency [94]. Quorum systems have been used as a foundation to implement various distributed systems and services, such as replicated databases [49, 61] and distributed read/write storages [14, 32, 94].

Distributed services can rely on a quorum system to achieve consistency and fault-tolerance through replication. That is, one purpose of quorum systems is to guarantee *consistency* in distributed computing with the aid of their key property, i. e., non-empty pair-wise intersections of quorums. Another important aspect of quorum systems is to achieve the goal of *fault-tolerance* in distributed systems. In a system with N faulty processes, a quorum is any *majority* of processes (any set of more than $N/2$ processes), and if there are $f < N/2$ processes failed by crashing, then at least

one quorum of uncrashed processes always ensures fault-tolerance in such a system. Fig. 1.1 illustrates a simple example of a read/write quorum system with majority quorums. There is a set of replica nodes $\{a, b, c, d, e\}$. When a client wants to access them by performing write/read operations, it only needs to contact a majority quorum which is one of the subsets of nodes, i. e., $\{a, b, c\}$ or $\{c, d, e\}$, intersecting with node c. In this way, the system can still provide service despite the failure of an individual replica node. In a system having arbitrary-fault processes, e. g., in order to tolerate f faults, a Byzantine quorum [94] is a set of more than $(N + f)/2$ processes, and there are two Byzantine quorums always overlapping in at least one correct process.

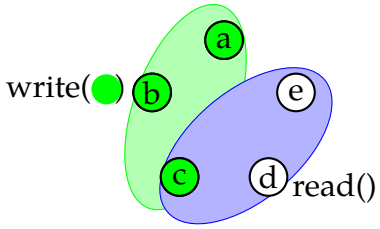


Fig. 1.1: Read/write majority quorums

1.1.3 *The Gorums Framework*

Commonly, replicated services can be implemented with distributed algorithms which rely on a quorum system [126, 127] to achieve fault tolerance. As explained above, given a quorum system, a process only needs to contact a quorum, e. g., a majority of the processes, to access the replicated state. In this way, a system can provide service despite the failure of some processes. However, communicating with and handling replies from sets of processes often complicate the protocol implementations. Gorums [87] is a recently proposed framework for implementing quorum-based distributed systems. It has been developed to alleviate the development effort for building advanced distributed protocols, such as Paxos [97] and distributed storage [14]. One of our research goals in this thesis is to provide a model-based testing approach for generating test cases to validate the correctness of the Gorums framework implementation and distributed systems and protocols implemented with Gorums.

1.1.3.1 *Gorums Abstractions*

The Gorums framework reduces the complexity of implementing quorum-based distributed systems by providing two core abstractions: a quorum call abstraction and a quorum function abstraction. The quorum call abstraction is used to invoke a set of remote procedure calls (RPCs) on a group of processes and to collect their responses. The remote procedure calls are based on the gRPC library developed by Google [50]. The quorum function abstraction can process responses to determine if a quorum has been obtained. These abstractions help to simplify the main control flow of protocol implementations.

Fig. 1.2 illustrates the interplay between the main abstractions provided by Gorums. Gorums allows clients to invoke a quorum call, i. e., a set of gRPCs, on a group of servers, and to collect their replies. The replies are processed by a quorum function

to determine if a quorum has been obtained. The quorum function is invoked every time when a new reply is received by the client, to evaluate whether the set of replies received so far constitutes a quorum or not.

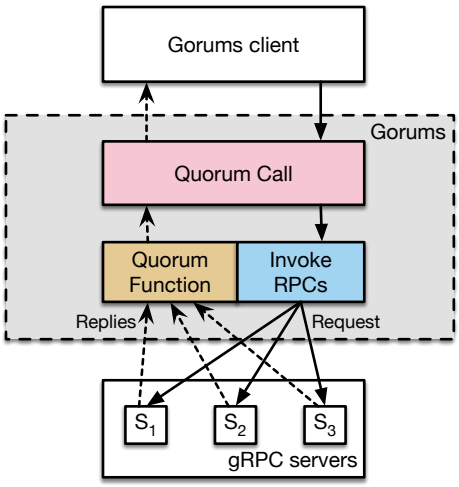


Fig. 1.2: Gorums architecture.

Given the Gorums framework, developers can specify several RPC service methods using `protobuf` [51], and from the specification, Gorums' code generator will produce code to facilitate quorum calls and collection of replies. Each quorum call method must provide a user-defined quorum function that Gorums will invoke to determine if a quorum has been obtained for that specific quorum call. Also, the quorum function will provide a single reply value, based on coalescing the reply values received from the different server replicas. This coalesced reply value is then returned to the client as the result of the quorum call. In this way, the invoking client does not see the individual replies.

1.2 Software Testing

The expectations of users and industry on high-quality software and software-controlled systems are growing rapidly. This is especially true as the development of software systems is becoming more complex. Testing is the primary approach that the industry uses to assess and evaluate the quality of software systems and to uncover problems during development. The aim of this section is to provide background on basic testing concepts which can be used to design test cases for software systems, including distributed systems.

1.2.1 Testing Levels

One of the key concepts of testing is the classification of levels based on testing activities in the software development life-cycle. These testing levels include *system testing*, *integration testing*, and *unit testing* [70]. The testing levels correspond directly to the design levels of software development. The model illustrating the relation between testing levels and design levels is known as the *V-Model* [70] and is shown

in Fig. 1.3. System testing assesses software concerning the requirements specification phase of software development. It is designed to determine whether the implemented software systems meet the requirement specification, and this level aims at checking if the system works as a whole and identifying design and specification problems. Integration testing evaluates software associated with the preliminary design phase. It is used to expose defects in the interfaces between integrated components (subsystems) of the system. Unit testing usually verifies the functionality of a specific program unit or section of code in the implementation.

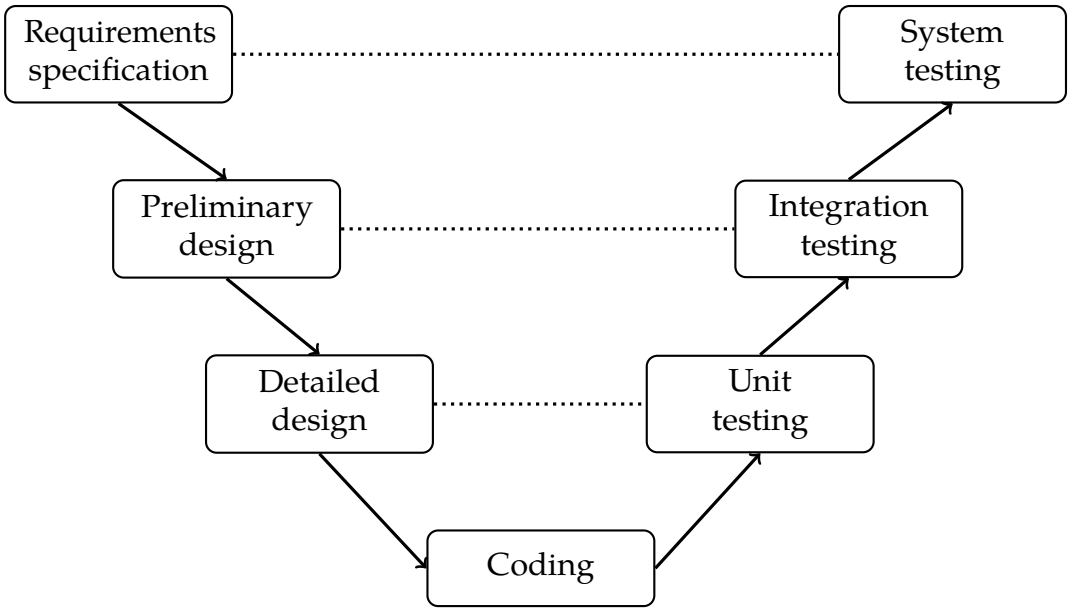


Fig. 1.3: Testing levels and design levels - the "V-Model" [70].

1.2.2 *Testing Terminology and Artifacts*

The terms and artifacts presented in this section are important in software testing, and these artifacts will also be used later in this thesis. Most of them are summarized below based on the IEEE Standard Glossary of Software Engineering Terminology [3].

For software testing, one of the important distinctions that needs to be understood is the definitions of a software *error*, *fault*, and *failure*. An error is an incorrect result produced by a human action, which can be considered as a mistake made by people while coding. A fault is an incorrect step, process, or data definition in a computer program. It is the representation of an error. A failure occurs when the code corresponding to a fault executes and indicates the inability of a system or component to perform its task according to specified performance requirements.

Another important distinction which needs to be made is between *validation* and *verification*. Validation is the process to evaluate software, a system, or components at the end of the development phase to ensure that it satisfies specified user requirements. In contrast, verification is the process to determine if software, a system, or component of a given development phase fulfills the requirements established during the previous phase.

When performing testing, tests usually include more than just input values to a software system under test. They consist of multiple testing artifacts. Specifically, a *test suite* is usually used to provide a finite set of *test cases*. Each test case consists of a finite set of *test inputs* and expected test output. The latter is known as *test oracles*. These test cases can be written manually or generated automatically. Then, given a test suite, a *test script* can be used to give a sequence of instructions for the execution of a test suite and to perform testing. Sometimes, a *test adapter* is used as the environment around a software system under test to provide test inputs via *test drivers*, observe *test outputs*, and compare the output to the test oracles.

1.2.3 Testing Approaches

Software testing can be classified based on testing approaches, and includes *black-box*, *white-box*, and *gray-box* testing. For the black-box testing approach [8, 70, 88, 139], a software system is considered as a black box without peering into its internal structure. The internal structure (or source code) of a software system is hidden, and the visible parts of this approach are the possible input and output values of test cases for the system. Also, the black-box testing approach is often called functional testing, since it only allows to test input-output functionality. The disadvantage of this approach is the lack of internal information for testing.

In contrast, the white-box testing approach [8, 70, 88, 139] tests the internal structure or source code of a software system since the internal information which can be used to create tests is visible to testers. For the sake of detecting problems of a software system, this approach uses test cases to execute and analyze specific parts of the source code or internal structure. Testing techniques for this approach includes fault injection and mutation testing. This approach is also called structure testing since it can access the structure of a software system. The disadvantage of this approach is the effort used to inspect the internal aspects of a program.

The gray-box testing approach [88, 139] is a hybrid form combining white-box and black-box testing. This approach aims to find defects by considering both the internal information and the input-output functionality.

1.2.4 Testing Coverage

Several basic testing coverage criteria have been proposed in the literature, including *statement coverage*, *branch coverage*, *path coverage*, and *modified condition/decision coverage (MC/DC)* [31, 48, 86, 141].

Statement coverage is the simplest coverage criterion, and measures the percentage of executed statements [90, 141]. A generated test suite that executes every statement in the program provides full statement coverage of the program. The percentage of the statements exercised by a test suite is a measurement of adequacy. Statement coverage is easy to measure, but is not sensitive to control structure that many programming errors are typically related to [90].

A generated test suite that exercises every control transfer branch in the program provides branch coverage [141]. The percentage of exercised branches during testing is also a measurement of the test adequacy of a test suite according to the branch coverage

criterion. However, branch coverage still has limitations as it only considers one branch at a time during testing [90].

Path coverage is a stronger measurement than statement coverage and branch coverage. It is concerned with a sequence of branch decisions (or statements) instead of only one branch (or statement) at a time. Path coverage considers combinations of branch decisions (or statements) with other branch decisions (or statements), which may not have been tested according to plain branch or statement coverage [90]. It is challenging to reach 100% path coverage, since the number of execution paths usually increases exponentially with each additional branch, or increases the number of cycles that cause infinitely many paths [86].

For modified condition/decision coverage (MC/DC) [31, 60, 109], each condition (Boolean expression) within a decision of a program must have taken all possible outcomes at least once when the program is exercised by a generated test suite. MC/DC addresses testing of boolean expressions and can be used to guide the selection of test suites. It has the requirement that each condition within a decision must be demonstrated to independently affect the outcome of the decision [31]. MC/DC cannot ensure the coverage of all conditions because some conditions in a decision may be masked by other conditions.

1.3 Model-based Software Testing

In this thesis, the main research focus is on *Model-based Software Testing* using techniques from the software testing domain for the development of distributed systems and protocols. Below, we present the basic concepts of model-based software testing.

1.3.1 The Process of Model-based Testing

Model-based testing (MBT) is a powerful approach for testing software systems. It typically relies on processes and techniques for constructing an abstract test model of the system under test (SUT) and its environment. MBT makes it possible to automatically generate concrete test cases from the abstract model, and manually or automatically execute concrete test cases on the SUT. The goal of MBT is failure detection and validation of the SUT by capturing observable differences between the behavior of the software implementation and the intended behavior of the SUT.

The process of MBT and its main elements are shown in Fig. 1.4. MBT involves the following major activities [124] (the steps refer to the numbers in Fig. 1.4):

Step 1: A model of the SUT is built based on requirements or specifications. This model is often known as a *test model*. The test model needs to be simpler and more abstract than the SUT so that it can be easily validated, modified, and maintained.

Step 2: Test selection criteria are chosen and testing strategies and techniques may be applied to guide the automatic test case generation. These criteria may include coverage criteria, data coverage heuristics, or stochastic characterizations.

Step 3: Test selection criteria are transformed into test case specifications that formalize the notion of the criteria to be used. These specifications are high-level descriptions

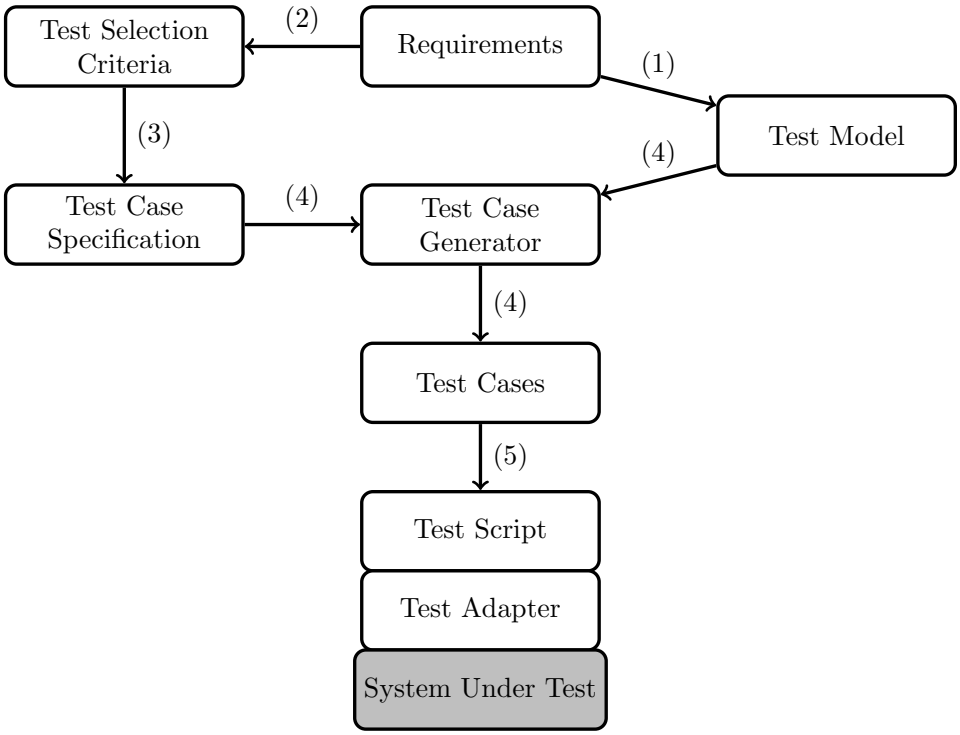


Fig. 1.4: The process of model-based testing [124].

of desired test cases such that an automatic test case generator can be used to derive a test suite.

Step 4: A test suite is generated consisting of a set of test cases that satisfy the test case specifications. In some cases, the test case generator needs to generate a small number of test cases that cover a large number of the test case specifications.

Step 5: Generated test cases are executed by test case execution, which can be performed with either a manual or an automated approach. During test case execution, a so-called *adapter* is used to inject test inputs into the SUT, collect test outputs of the SUT, and finally compare test outputs against the expected outputs (called test *oracles*). The adapter can be implemented either as a separate software component or integrated within a *test script*. A test script is some executable code usually used to perform test case execution.

1.3.2 *A Taxonomy of Model-based Testing*

Utting et al. [125] have proposed a taxonomy of MBT categorized into three main categories and along six dimensions. Fig. 1.5 illustrates these main categories including the categories of *model specification*, *test generation*, and *test execution*.

Model Specification. The *model specification* category has a strong connection with step 1 of MBT, and it has three dimensions, consisting of *model scope*, *model characteristics*, and *model paradigm*.

A model is an abstract representation of the SUT. The model scope is classified into a binary decision depending on whether the model specifies only the inputs to the

SUT or specifies both inputs and outputs of the SUT. The input models of MBT have the disadvantage that the generated tests will not be used as oracles, and hence they cannot check the correctness of the output values from the SUT. On the contrary, the input-output models of the SUT are able to send inputs into the SUT and capture some of the intended behavior of the SUT.

The model characteristics involve timing issues, nondeterminism, and the continuous or discrete nature of the model. Timing issues are often related to real-time systems. This means that if the SUT is a real-time system, then real-time constraints need to be considered and tested. Real-time systems are known to be difficult to test, and in this case, the timed models (models with timing annotations) are usually used to represent and test such systems. Given the inputs of the SUT, the outputs of the SUT can be deterministic. Therefore, a model of the SUT can also be deterministic. However, the SUT sometimes has concurrency issues due to internal parallelism, and hence there might be some alternative generated outputs. In terms of dynamics, systems can have discrete behaviors, continuous behaviors or a mixture of both behaviors. The latter is known as a hybrid system. Therefore, the models representing such systems can also be discrete, continuous or hybrid.

The model paradigm dimension concerns the notations that are used to describe the model of the SUT. Different notations for modeling used in MBT can be classified including state-based notations, transition-based notations, history-based notations, functional notations, operational notations, stochastic notations, and data-flow notations [84, 125].

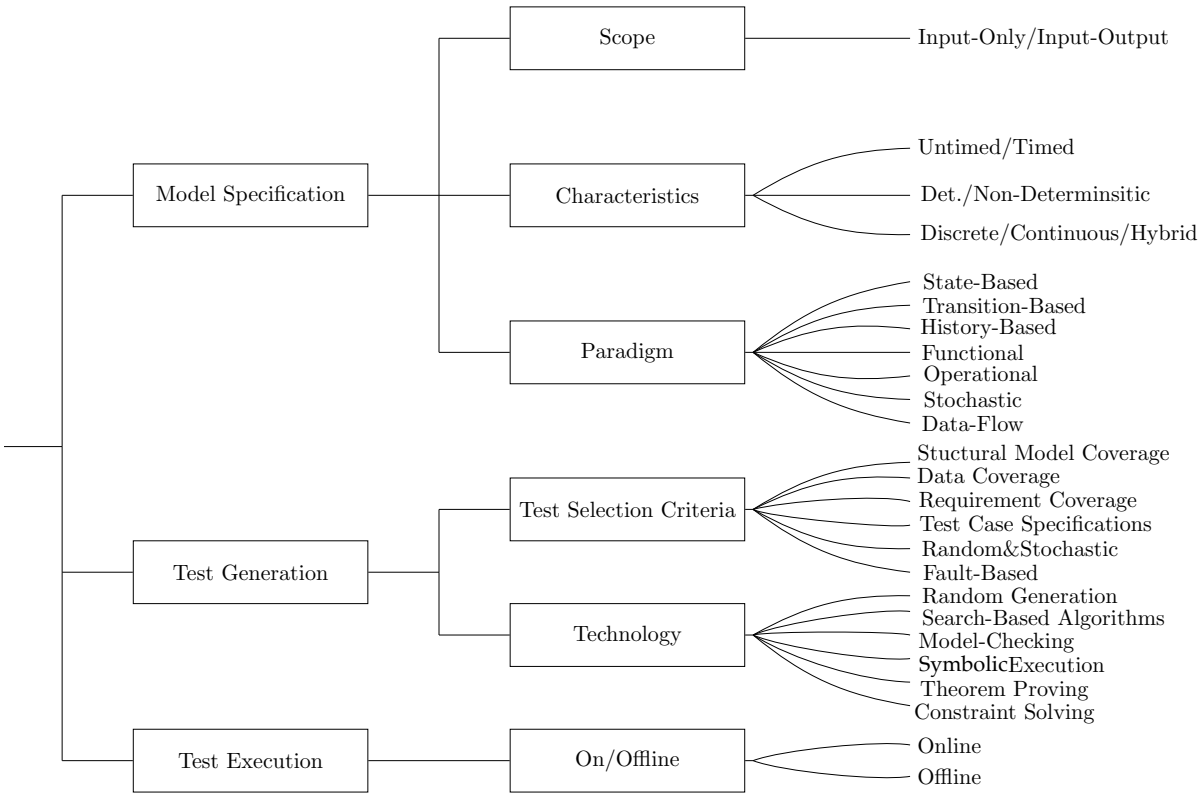


Fig. 1.5: A taxonomy of model-based testing approaches [125].

Test Generation. The *test generation* category is primarily concerned with *test selection criteria* and *test generation technology*. The test selection criteria determine how tests are selected and generated in steps 2 and 3 of the MBT process. In general, it is not possible to define a best criterion, so various test selection criteria can be considered for MBT. These criteria are listed and introduced briefly below, but discussed in detail by Utting et al. [125]. They include *structural model coverage*, *data coverage*, *requirements coverage*, *test case specifications*, *random and stochastic coverage*, and *fault-based coverage*.

Structural model coverage criteria specify the selection of tests based on the use of the structure of the model. This may involve structural elements such as nodes and arcs from transition-based models, or conditional statements within models. For transition-based models, graph coverage criteria can be used to control the test generation. For conditional statements within models, code-based structural coverage criteria can be useful for complex boolean decisions. Data coverage criteria focus on selecting a small number of test values from a huge data space. For these criteria, to compare with random testing, boundary analysis and domain analysis are considered as techniques involving fault detection heuristics for test generation.

Requirements-based coverage criteria are based on the informal requirements of the SUT. This means that the elements of the model, such as transitions of a state machine are related to the informal requirements of the system. Therefore, coverage can be applied to these requirements, and test case generation can ensure that these requirements are covered. Explicit test case specifications can also obviously be used to control test generation. The test engineers are involved in writing test case specifications in some formal notation which is then used to select tests to be generated. For instance, some execution paths through the model may be the only focus for testing, or frequently used scenarios and their particular paths are in focus and ensured to be tested. Commonly used notations include UML [70], FSMs [70], temporal logic formula [16], and Markov chains [16] for expressing test objectives. Random and stochastic criteria are mostly applicable to environment models, because it is the environment that determines the usage patterns of the SUT. Fault-based criteria are commonly used to find faults in the SUT. An example of these criteria is mutation coverage which involves mutating the model of the SUT.

Given the test model and some test case specifications, test cases can be automatically generated using techniques such as *random generation*, *search-based algorithms*, *(bounded) model checking*, *symbolic execution*, *theorem proving*, and *constraint solving*. Random generation usually performs a random walk on the test model to generate test cases. The search-based techniques rely on heuristic search or evolutionary algorithms for test case generation. (Bounded) model checking uses model checkers to generate test cases (traces) based on properties of a system for verification. Symbolic execution uses symbolic traces as sets of input values, generated by executing a model of the SUT. Theorem proving is used to check the feasibility of formulas appearing as the guards of transitions in state-based models. Finally, constraint solving uses constraint solvers to select data values from complex domains for test case generation. In general, several of these techniques are often used in combination to complete the difficult task of automated test generation from a test model.

Test Execution. In online testing, the test generation algorithms not only provide inputs to the SUT, but can respond to the output of the SUT. This kind of testing is commonly performed when the SUT is non-deterministic, and is also known as *on-the-fly* testing. In *offline* testing, tests are generated before test execution. Unlike online testing, it is more difficult for offline test generation to generate test cases from the non-deterministic test models. However, the advantage of offline test generation is that once tests have been generated, they can be reused many times on the SUT in the future. Test execution can also be performed on different machines or in a variety of environments. The generated test cases may be executed manually or automatically. Automatic execution may require more work, for example, to develop an adapter program (or the program to generate this adapter) to read test values and execute tests against the SUT automatically.

1.4 The Modbat Model-based API Tester

In this thesis, the model-based testing tool *Modbat* [10] has been applied and extended as part of our research into MBT. Recently, Modbat has been used successfully for testing the ZooKeeper [64] distributed coordination service by exploring the interleavings and non-deterministic outcomes caused by scheduling decisions and network communication [11].

According to Utting et al. [125], many commercial and academic MBT tools fit into the taxonomy in Fig. 1.5. For the model scope of the taxonomy in Fig. 1.5, a Modbat model can specify both inputs and outputs of the SUT. For model characteristics, transitions of Modbat support non-deterministic outcomes. In addition, Modbat aims at performing online testing of state-based systems [10].

Modbat uses *extended finite state machines* (EFSMs) [28] to express test models in a domain-specific language based on Scala [108], and can explore the transition system and execute the functions specified on the transitions. Test case generation in Modbat uses a random based search, and state- and transition coverage as metrics. In this thesis, the test case generation approach of Modbat has been extended with a search-based approach, and path coverage has been implemented as one of the test criteria for Modbat.

Fig. 1.6 (left) illustrates the ChooseTest model of Modbat as a simple example. It consists of three states: `"ok"`, `"end"`, and `"err"`. Transitions are declared with a concise syntax: `"origin"` → `"dest"` := {`action`}. A valid execution path in a Modbat model starts from the initial state (automatically derived from the first declared state) and consists of a sequence of transitions.

Modbat has built-in *require* and *assert* methods with associated actions. The *require* methods in transitions check if preconditions are fulfilled. Preconditions must be fulfilled in order for a transition to be enabled. The *assert* methods in transitions are used as assertions to check if specific conditions are fulfilled. The ChooseTest model in Fig. 1.6 uses *require* in the action part as a precondition to check if a call to the random function *choose* returns 0 (10% chance). Only in that case is the transition from `"ok"` to `"err"` enabled. Function *assert* is then used to check if a call to *choose* returns non-zero. If 0 is returned (10% chance), the assertion fails. Thus, transition "ok" → "err" is rarely enabled; and if enabled, it fails only infrequently.

```
class ChooseTest extends Model {
  "ok" -> "ok" := skip
  "ok" -> "end" := skip
  "ok" -> "err" := {
    require(choose(0, 10) == 0)
    assert(choose(0, 10) != 0)
  }
}
```

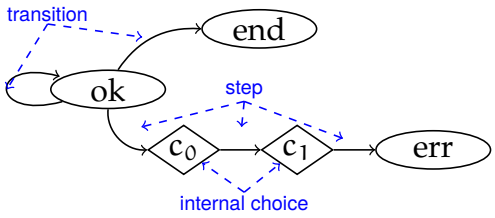


Fig. 1.6: Model ChooseTest (left) with steps and internal choices (right).

Modbat supports two kinds of *choices*: (1) the choice of the next transition is available before the current transition is executed; (2) within an action, choices can be made based on the parameters used as input to the SUT or used for computations inside the actions. The latter are called *transition internal choices*, which can be choices over, e. g., a finite set of numbers. These choices are obtained in Modbat by calling the function *choose*. In our example, the action in transition `"ok"` to `"err"` has two transition internal choices shown as c_0 and c_1 in Fig. 1.6 (right). Note that we only show the successful transition for the transition internal choices. In order to distinguish these two kinds of choices, Modbat divides an action into smaller *steps* (shown in Fig. 1.6 (right)).

1.5 Coloured Petri Nets

In this thesis, *Coloured Petri Nets (CPNs)* [67] and *CPN Tools* [37] are used to support the MBT approach for distributed systems and protocols. CPNs is a graphical language for modeling and validation of systems where concurrency, communication, and synchronization constitute the key aspects. CPNs combines *Petri Nets* [110] and the functional programming language CPN ML which is based on *Standard ML* [120]. Petri Nets provides the foundation of the graphical notation and the primitives for modeling concurrency and communication while Standard ML is used for modeling data. A CPN model of a system represents both the states of the system and the transitions causing state changes of the system.

Construction and analysis of CPN models are supported by CPN Tools which has been widely used for modeling and verifying models of complex distributed systems. CPNs has been applied to many domains including communication protocols [19], data networks [20], distributed algorithms [111], and embedded systems [5]. Recently, work on automated code generation with CPNs and CPN Tools has also been done [75].

Given CPNs and CPN Tools, it is possible to investigate different scenarios and explore behaviors of the modeled system using simulation-based analysis and verify behavioral properties using state space methods and model checking. Simulation-based analysis with CPNs and CPN Tools aims at debugging and investigating the system design. The simulation of a CPN model can be performed in an interactive or automatic way with CPN Tools. An interactive simulation provides a way to execute a CPN model based on steps (similar to single-step debugging); an automatic simulation allows to execute a CPN model in the same way as a program execution. State space methods can be used to verify system properties with the help of state-space exploration. The basic idea of state-space exploration is to compute all reachable states and states changes

(caused by occurring transitions) of the CPN model and represent them as a directed graph. In such a directed graph, the nodes represent states and the arcs represent occurring transitions. State-space exploration can be performed fully automatically, and it can help to investigate different verification questions related to the behaviors of the system. However, if the CPN model is too complex, it may suffer from the state space explosion problem.

A CPN model is organized as a set of *modules*. A CPN model contains *places* (drawn as ellipses or circles), *transitions* (drawn as rectangular boxes), a number of directed *arcs* connecting places and transitions, and finally textual *inscriptions* next to the places, transitions, and arcs. A module of a CPN model can have *substitution transitions* (drawn as rectangular boxes with double lines). The basic idea of hierarchical CPN models is to associate a module with each substitution transition. When a module is associated with a substitution transition it is said to be a *submodule* [68]. The name of the submodule is shown in a name-tag next to its associated substitution transitions.

Below, we use a CPN model of a *Two-phase Commit (2PC)* protocol [52, 53] to briefly explain how a CPN model is represented with CPN Tools. The 2PC protocol operates in two communication phases. The first phase involves a process, known as a *coordinator*, which propose a value to every worker (processes in the system) and gathers responses. Any process can act as the coordinator. The response of a worker is a *YES* vote if it accepts the proposed value, otherwise, the response is a *NO* vote. For the second phase, the coordinator sends the decision of the vote to the workers by using a *COMMIT* if they all voted *YES*, otherwise a *ABORT* is sent to all workers to abort the protocol. In a later chapter, the CPN model of the 2PC protocol is also used as an example for our MBT research on test case generation.

The CPN model for the 2PC protocol is comprised of four hierarchically organized modules. Fig. 1.7 shows the CPN module for the coordinator process of 2PC and Fig. 1.8 shows the CPN module for the worker processes. The top-level CPN module and the submodule of the CollectVotes substitution transition in Fig. 1.7 have been omitted here. In the coordinator module, each port place (ellipses drawn with a double border) is linked to the accordingly named place in the workers module by so-called port-socket assignments. The colour sets and the variables used are shown in Fig. 1.9.

The coordinator starts by sending a message to each worker (by enabling and occurrence of transition SendCanCommit), asking whether the transaction can be committed or not. Each worker can vote either *Yes* or *No* (by enabling and occurrence of transition ReceiveCanCommit). The coordinator then collects each vote via the CollectVotes submodule of the CollectVotes substitution transition. Then, the coordinator sends back either an abort or commit decision, based on the collected votes, and it will decide on commit if and only if all workers voted yes. The workers who voted yes then receive the decision (by enabling and occurrence of transition ReceiveDecision) and send back an acknowledgment to the coordinator. The coordinator then receives all acknowledgments (by enabling and occurrence of transition ReceiveAcknowledgement). After the execution of the model, a token with colour abort or commit will be placed on the place Completed depending on whether the transaction was to be committed or not.

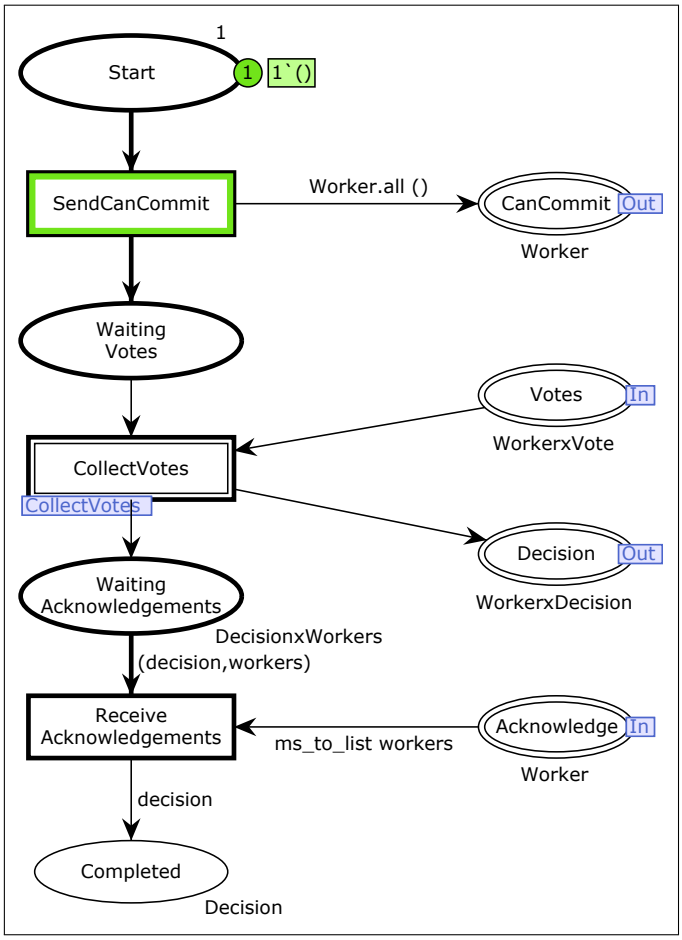


Fig. 1.7: Coordinator module of the 2PC CPN model.

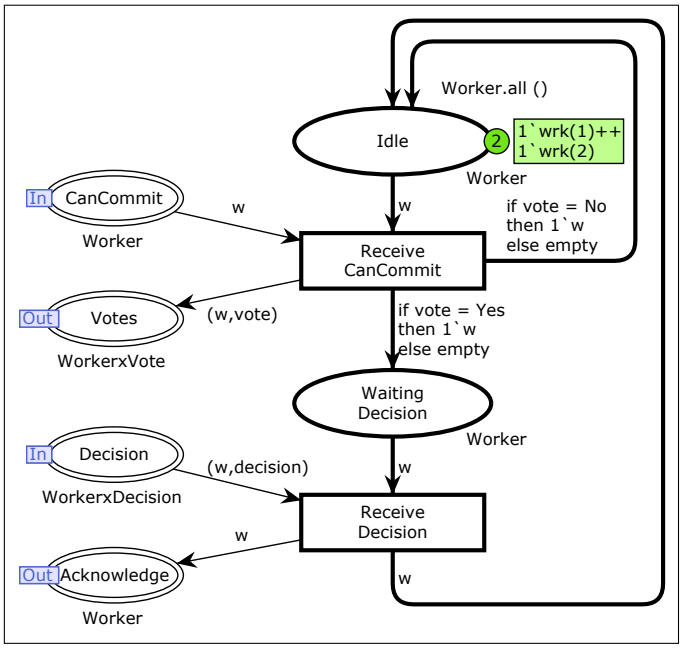


Fig. 1.8: Workers module of the 2PC CPN model.

```
val W = 2;
colset Worker  = index wrk with 1..W;    var w : Worker;
colset Workers = list Worker;            var workers : Workers;

colset Vote     = with Yes | No;         var vote : Vote;
colset Decision = with abort | commit;   var decision : Decision;

colset WorkerxVote      = product Worker * Vote;
colset WorkerxDecision  = product Worker * Decision;
```

Fig. 1.9: Colour set and variable declarations.

1.6 Research Questions

In this thesis, we focus on applying model-based software testing approaches and techniques for the development and testing of distributed systems and protocols. Specifically, our research focus is on the following research questions:

RQ1: *How can model-based testing be applied to detect errors and to ensure the correctness of quorum-based fault-tolerant distributed systems and protocols?*

RQ2: *How can Coloured Petri Nets and CPN Tools be used to support model-based testing for distributed systems and protocols?*

RQ3: *How can test criteria and test case generation technology of model-based testing measure test adequacy and effectively generate test cases?*

RQ1 is concerned with the investigation of MBT approaches and related software testing techniques for quorum-based fault-tolerant distributed systems. Distributed systems are notoriously difficult to implement correctly since it is challenging to cope with both concurrency and failures, e. g., due to crashes and network partitions. Thus, when designing and implementing distributed systems, it is important to ensure correctness and fault-tolerance. Distributed systems employ distributed protocols with complex logic to tolerate individual component failures without causing service disruption for users. However, these protocols, such as the Paxos consensus protocol [81, 82, 97], are also known for being difficult to understand and implement correctly.

The Gorums framework has abstractions to reduce the complexity of the implementation of quorum-based distributed systems and protocols. Especially, these abstractions help to simplify the main control flow of the protocol implementation. Therefore, the Gorums framework can be used as the foundation for implementing, quorum-based fault-tolerant distributed systems and protocols, which then can be used as the system under test. Also, the widespread adoption of the Gorums framework will depend on the correctness of its implementation. This has motivated us to test the Gorums middleware and provide an MBT approach that can be used to also test applications in general that rely on the Gorums framework.

We choose CPNs as a theoretical foundation for this research question because it has a strong track record for modeling distributed systems, and is able to create

parametric models, and perform model validation. CPNs also have mature tools to support both simulation and state space exploration, which is important for practical experiments and evaluation. Therefore, the goal of our research into **RQ1** is to provide a model-based testing approach for generating test cases from CPN models to validate the correctness of the Gorums framework implementation itself and distributed systems and protocols implemented with Gorums.

RQ2 aims at providing the supporting tools and techniques for MBT in the context of CPNs and CPN Tools. It is important to develop such software tools and techniques so that we can detect errors and ensure correct behavior and stable operation of distributed systems and protocols. CPNs and CPN Tools have been widely used for modeling, validation, and verification of software systems, and recently, work has been done with code generation from CPN models. However, applications of MBT via CPNs and CPN Tools have only been explored to a limited extent. Therefore, the aim underlying this research question is to implement software tools based on CPNs and CPN Tools to support MBT. In particular, this involves test case generation in order to test distributed systems and protocols implemented via the Gorums framework.

RQ3 focuses on the investigation of test criteria and test case generation technology of MBT, aiming at measuring test adequacy and generating test cases effectively, i. e., generate sufficiently good test cases in order to obtain the desired test results.

MBT is conducted via the automatic generation and execution of test cases. However, it is a challenge to generate sufficiently many and diverse test cases for a good coverage of the SUT, especially for complex distributed systems and protocols. Therefore, before the test case generation and execution of the MBT process, it is important to chose test adequacy criteria to measure and evaluate the extent to which sufficient test cases have been generated and executed against the SUT. Although MBT can automatically generate test cases from abstract (formal) models of the SUT, it is infeasible to explore and generate all the possible test cases for complex software systems. This means that a challenging decision needs to be made on how many test cases to generate. Another important challenge is to address the test adequacy criteria by generating a small test suite having few redundant test cases.

Uncontrolled random approaches cannot address these challenges of MBT. Our aim with this research question is therefore to provide an approach to measure test adequacy and propose an effective test case generation approach to generate sufficiently many and good test cases against the SUT in order to obtain better test results. For this research question, we use the Modbat tester as our foundation instead of CPNs. The reason is that Modbat has a suitable software architecture and benchmark suites to further develop test criteria and test case generation techniques.

The Modbat tester has a standard random search approach to generate test cases. However, the random approach might result in test suites having redundant test cases which only cover few execution paths of the Modbat models and the SUT. Therefore, we also aim at improving the test case generation in the Modbat tester, so that it can perform test case generation more effectively.

1.7 Research Method

Fig. 1.10 shows the research method and associated activities underlying our research into MBT for distributed systems and protocols. The research method focuses on three main areas including theoretical foundations and approaches, MBT software tools and techniques, and SUT case studies and experiments.

The process of the research method and associated activities is first to develop and propose our approaches for MBT, which can be used as the theoretical foundations. Then, based on theoretical foundations, we 1) implement MBT software tools and techniques; and 2) develop case studies including the implementation of the systems under test. After that, we apply the implemented software tools and techniques to perform MBT on the systems under test from the case studies. The results obtained from the experiments of the case studies then serve to evaluate the approaches we have proposed for MBT.

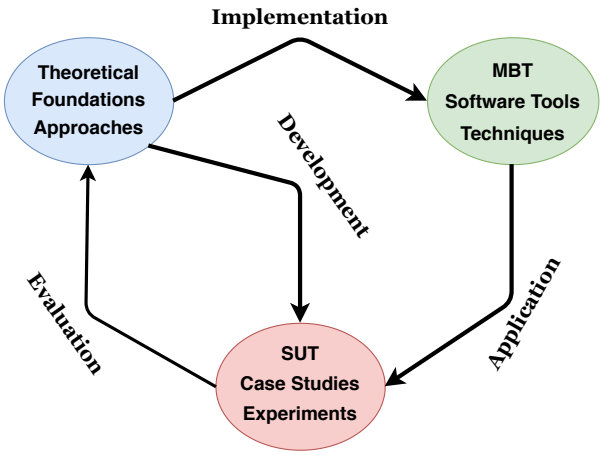


Fig. 1.10: Research method and activities.

1.8 Outline

This thesis is organized into two main parts. Part I gives an introduction to, and an overview of, the research field of this thesis, presents our research methodology, discusses the results obtained and our research contributions, and puts our work into a state-of-the-art context through the discussion of related work. Part II consists of a collection of four published and peer-reviewed articles [128, 132–134], and one submitted international conference paper [129].

The rest of Part I consists of chapters that introduce the main research topics and related work for each of the articles included in Part II, and is organized as follows:

Chapter 2:
Distributed Systems and Protocols.
This chapter discusses distributed systems and protocols. The chapter introduces the theoretical foundation for developing distributed systems and protocols as our

systems under test, so that MBT can be performed against them. Specifically, some basic concepts and abstractions for implementing distributed systems are presented. Distributed storage systems and distributed consensus algorithms and protocols are in focus of the discussion as they constitute the main foundation for the systems under test implemented in the case studies developed for this thesis.

Chapter 3:
Model-Based Testing for Fault-Tolerant Distributed Systems and Protocols.
This chapter discusses the research topic related to articles [132, 133] included in Part II. It focuses on the research into using Coloured Petri Nets (CPNs) for model-based testing of quorum-based distributed systems and protocols. A model-based testing approach with a supporting testing framework has been proposed, with a special focus on fault-tolerance of quorum-based distributed systems and protocols. Two case studies have been developed based on the proposed approach and testing framework, consisting of a distributed storage system and a singe-decree Paxos distributed consensus protocol.

Chapter 4:
A Software Tool for Test Cases Generation with Coloured Petri Nets.
This chapter discusses the MBT/CPN tool developed for the research on test case generation for model-based testing. The details of the tool are presented in the article [134] in Part II. The application of the tool has been demonstrated via model-based testing of an implementation of the 2PC protocol. In addition, the tool has been used to test the implementations of a distributed storage system and the Paxos distributed consensus protocol in [132, 133].

Chapter 5:
Path Coverage Visualization and Multi-objective Search with Modbat.
This chapter first introduces our approach to measure and visualize execution path coverage of test cases. The technique has been presented in the article [128] in Part II. This chapter also discusses our search-based algorithm proposed in the article [129], which we have developed to guide test case generation. The approach applies multi-objective optimization and a genetic algorithm to obtain optimal test cases. Both approaches have been developed by extending the Modbat tester and experimentally evaluated on a collection of examples, including the ZooKeeper distributed service [64].

Chapter 6:
Conclusions and Future Work.
This chapter re-visits our research questions and provides a summary of the main contribution of this thesis. We also outline several future research directions of model-based software testing for distributed systems and protocols, based on the work undertaken for this thesis.

1.9 Supplementary Material

In addition to the articles [128, 129, 132–134] included in Part II of this thesis, two workshop articles have been published presenting initial research results:

[130] R. Wang, L. M. Kristensen, H. Meling, and V. Stolz. Application of Model-based Testing on a Quorum-based Distributed Storage. In CEUR Workshop Proceedings, Petri Nets and Software Engineering (PNSE'17), volume 1846, pages 177–196, 2017.

[131] R. Wang, L. M. Kristensen, H. Meling, and V. Stolz. Model-based Testing of the Gorums Framework for Fault-tolerant Distributed Systems. In Proceedings of the 29th Nordic Workshop on Programming Theory (NWPT), Turku Center for Computer Science, Finland, 2017.

Two additional articles have been published on topics that are related to the research questions addressed in this thesis:

[93] F. Macias, T. Scheffel, M. Schmitz, and R. Wang, M. Leucker, A. Rutle, V. Stolz. Integration of Runtime Verification into Metamodeling. In Proceedings of the 28th Nordic Workshop on Programming Theory (NWPT). Aalborg University, Denmark. 2016.

[92] F. Macias, T. Scheffel, M. Schmitz, and R. Wang. Integration of Runtime Verification into Metamodeling for Simulation and Code Generation (position paper). In Runtime Verification, volume 10012 of Lecture Notes in Computer Science, pages 454–461. Springer International Publishing, 2016.

The papers [92, 93, 130, 131] are not formally part of this thesis and will hence not be discussed further.

All the models, distributed protocols and tools implemented for the publications included in this thesis can be found on Github [4].

CHAPTER 2

DISTRIBUTED SYSTEMS AND PROTOCOLS

In this chapter, we provide background information on application areas of distributed systems and basic concepts and abstractions that are being used for the development of distributed systems and protocols. This is followed by a more detailed discussion of distributed consensus algorithms and protocols as they are the focus of our research into MBT. We have chosen some typical distributed systems and protocols from this chapter as representatives to implement and use as the systems under test in our evaluation case studies to be presented in later chapters.

2.1 Distributed Applications

For distributed systems, the underlying physical system is mainly represented by *processes* and *links*. A process may represent a computer, a processor within a computer, or a specific thread of execution within a processor. Links represent the physical or logical network that enables communication among processes. Processes then cooperate and exchange messages using the network provided by communication links.

Many applications such as the Web or E-mail services rely on the simplest form of distributed computing: the *client-server* paradigm. The *server*, a centralized process, provides a service to many remote *clients*. The communication between the clients and the server usually follows a request-response pattern of interaction. However, there are often not only two but several processes that need to cooperate and synchronize to achieve a common goal. Examples of such applications, discussed briefly here, include information dissemination engines, process control systems, cooperative systems, distributed databases, and distributed storage systems [25].

For *information dissemination* in distributed applications, processes may play one of two roles: information producers (publishers) or information consumers (subscribers). Publishers produce information in the form of notifications and subscribers register their interest in receiving notifications. Applications for *process control systems* involve controlling the execution of a physical activity through several software processes. Examples of applications with process control systems are controlling the dynamic location of a train or an aircraft, controlling the temperature of a house, or controlling the automation of a robotic production plant. In these cases, several of the processes are connected to sensors. Then, the processes might need to exchange sensor values and output some common values, e. g., print a single location of the train on the driver control screen. Different local sensors controlled by associated processes may observe

slightly different input values due to inaccuracy or failure. However, the cooperation of these processes should be achieved even if some sensors has crashed or not observed anything. This requirement can be captured if all processes agree on the same set of input values for the control algorithm, i. e., all processes reach the *consensus*.

For applications involving *cooperative work*, an example is a shared network that may have different nodes for users to cooperate on a common document such as in Google Docs, or setting up a distributed dialogue for an online chat or conference. Such cooperation usually relies on a distributed shared memory, typically accessed through *read* and *write* operations by the users to store and exchange information.

For a distributed transaction of *distributed databases*, database servers need to coordinate their activities and decide whether to commit or abort transactions. For example, they might decide to abort the transaction if one server detected a concurrency control inconsistency or the crash of some other server. This means that distributed databases need to ensure that all transaction managers have a consistent view of the running transactions and make consistent decisions on how these transactions should be serialized.

A *distributed storage* system has storage nodes for the distribution of data. Each storage node provides a small portion of the overall storage space. Since a single data item may be stored over several nodes, it is common to contact multiple nodes to access the stored data. Such a storage system relies on distributed data to increase the fault-tolerance of the overall system and for reducing the load on each storage node.

2.2 Synchronous and Asynchronous Systems

For asynchronous distributed systems, we do not assume that processes have access to any sort of a shared physical clock. The passage of time can be measured with a logical clock according to the transmission and delivery of messages. The time measured this way is known as logical time. However, in an asynchronous system, there is no fixed upper bound on the time required for a message to be sent from one processor to another [42]. The *consensus* problem in distributed computing is hard to solve in an asynchronous system, since no completely asynchronous consensus protocol can tolerate even a single process death. Such a process death can then cause any distributed protocol to fail in reaching an agreement [46].

In contrast to asynchronous systems, in a synchronous system, a known fixed upper bound on the time is required for a message to be sent from one process to another and there is a known upper bound on the relative speeds of different processes [42]. Also in synchronous distributed systems, several useful services can be provided, such as the *timed failure detection*, *measure of transit delays* and *synchronized clocks* [25].

The concept of partially synchronous [42] in a distributed system has also been proposed. It lies between synchronous and asynchronous systems, which means that the system may not always behave synchronously, but there are periods where the system operates synchronously long enough to do useful jobs such as executing an algorithm or terminating it.

2.3 Distributed Programming Abstractions

Distributed programming abstractions are the basic elements used to define a *distributed-system model*, and a distributed-system model usually combines (1) a process abstraction; (2) a communication link abstraction; (3) a failure detection abstraction; and (4) a leader election abstraction [25]. In the following, we discuss the important abstractions that constitute a *distributed-system model*.

2.3.1 Process Abstractions

Distributed programming abstractions typically consist of a collection of software components, at least one for each process, that are intended to satisfy some common properties. A process executing a distributed algorithm fails if it does not behave according to the algorithm. If a failure occurs, then all components of this process fail at the same time. Process abstractions can be classified based on the nature of the faults causing processes to fail. Such failures could involve, e. g., a crash or arbitrary and even adversarial behavior. Here, three kinds of process abstractions will be briefly discussed, including the *crash-stop*, *crash-recovery*, and *Byzantine* process abstractions.

The *crash-stop* process [46, 56, 65, 116] abstraction means that the process executes correctly until some time t (crash at time t), and it never recovers after that time. This is called a *crash fault*. This abstraction indicates that the distributed system or algorithm cannot rely on such processes to recover after the crash.

Sometimes, it is too strong to assume for certain distributed environments that some particular processes crash and never recover. Therefore, instead of the crash-stop process abstraction, the *crash-recovery* process [6, 56, 65, 103] abstraction is considered. For this abstraction, a process is faulty if it either crashes and never recovers or if it keeps crashing and recovering infinitely. Otherwise, the process is considered to be correct. In this case, a process that crashes and then recovers (a finite number of times) is correct. However, after the recovery, the process might send new messages that conflict with messages sent by the process before the crash. Therefore, each process needs to have stable storage (a log) to cope with problems due to the crash and recovery. This can be done by storing messages already received in the stable storage, and this stable storage can be accessed by store and retrieve operations.

A process may fail in an arbitrary manner and such failures are known as *Byzantine* failures [40, 71, 72, 83]. Arbitrary faults are the most complex ones to tolerate, but it is the only acceptable option when unknown or unpredictable faults may occur. An arbitrary fault could be either intentional, malicious, or an error in the implementation, the programming language, or the compiler. For a process that has such arbitrary-fault behavior, the *Byzantine* process abstraction can be used to model it.

2.3.2 Communication Link Abstractions

The network components of the distributed system are represented by communication link abstractions. These abstractions may be implemented by different topologies which provide full connectivity among the processes, with each pair of processes connected by a bidirectional communication link. Two abstractions are *point-to-point* communication abstractions supporting interaction between pairs of processes,

and *broadcast* communication abstractions providing connections from one to many processes [25]. For the communication link abstractions, messages exchanged between processes are unique and have sufficient information for the recipients to identify their senders [25]. If the messages are exchanged in a request-reply manner, then the processes also need to identify which reply message is a response to which request message with the help of timestamps or sequence numbers of messages.

Moreover, given that physical communication links may be partitioned, and that they may also lose, duplicate and modify messages, it is unrealistic to assume that communication links are perfect. Therefore, in a distributed system, when transmitting a message through the network, it is possible for the messages to be lost, duplicated, or even modified and distorted. For the unreliability of such a network, communication link abstractions can use e. g., retransmission of messages until they reach their recipients and message integrity checks to recover from the vulnerability of the links.

In traditional distributed applications such as the classic *client-server* scheme, communication is often established between two processes. For example, a server process provides a reply to clients for a request they have sent to the server. *Point-to-point* communication abstractions are useful to support such an interaction scheme between a server and clients [25]. It is also helpful for such applications that the point-to-point communication is reliable. Typical protocols for point-to-point communication are reliable transport protocols, such as TCP. With reliable point-to-point communication, applications are free from handling issues such as message loss, duplication, or acknowledgments between the involved processes.

As distributed applications become more complex and more processes are involved in the interactions in a coordinated manner, the *broadcast* communication abstractions are convenient to use to disseminate information among a set of processes. It allows a process to send a message to a *group* of processes and ensures that the processes have an agreement on the messages they receive [25]. Furthermore, reliable broadcast communication abstractions ensure that the messages received by each receiver follow the same order and achieve a form of consistency.

2.3.3 *Failure Detection*

For failure detection, the *distributed failure detector* abstraction [6, 27, 40, 58, 71, 72] provides information about which processes have crashed and which are correct. For the crash-stop process abstraction in synchronous systems, for instance, a failure detector for crash faults provides an accurate failure signal in case a remote process stops behaving properly, and crashes can be accurately detected by a failure detector using *timeouts*. It means, for instance, that a sender process can detect a crash of the receiver process when there is no response from the receiver process to the sender process within the timeout period.

2.3.4 *Leader Election*

The *leader election* abstraction [57, 85, 95, 99] is used to identify a process that has not failed. This process may then act as the leader of the other processes. Generally, the leader election abstraction has the task to choose one process as a leader of the group

of processes in the system, and a new leader should be elected if the current leader crashes. Leader election is, for example, useful for a set of replica processes within consensus algorithms to coordinate their activities to provide high availability of the service which tolerates the failure of some processes. It works by electing one correct process as a leader of the replica processes, and the other processes can then be updated by the leader. If the leader crashes, one of the other replica processes is elected as the new leader.

2.4 Distributed Storage Systems with Shared Memory

Given a set of processes that communicate by sending messages to each other over a network, a distributed storage system can be implemented based on the notion of a *register* abstraction [80] that results in emulation of shared memory. In other words, the algorithms used to implement register abstractions are inspired directly from the implementation of a distributed storage system. The processes in the system use the registers to communicate with each other and store information.

Here, we consider the behavior of registers accessed concurrently by multiple processes. These registers store values and are accessible by processes through two operations, *read* and *write*. A read operation is invoked by a process to access the register abstraction that has a value stored, and then this read operation returns the value stored to this process. A write operation is invoked by a process to update the stored value in the register abstraction, and it returns an acknowledgment to the process indicating that the value is updated in the register abstraction. Each correct process invokes read/write operations on a register in a sequential manner. A register is usually initialized to a special value by some write operation. Each value written to a particular register is assumed to be unique, implemented by adding a unique timestamp given by the process to the value written.

Register abstractions can be distinguished based on the set of processes that may perform read/write operations on a register:

***single-writer, single-reader* register [80]:** a register with only one writer and one reader, also known as a $(1, 1)$ register for short, where a writer and a reader are both specific processes, respectively.

***single-writer, multi-reader* register [80]:** a register with only one writer but N readers, also called a $(1, N)$ register, which means that any process can read from the register.

***multi-writer, multi-reader* register [118]:** a register to which every process may have access by read/write operations, also called a (N, N) register.

The processes that access a register might fail, e. g., by crashing, and such a crash is unpredictable. This means that a process invoking a read/write operation on a register may not have time to finish the operation due to failures. This situation makes distributed computing challenging due to the non-determinism in distributed storage systems. For an example, if a writer is executing to access the register, and before it completes, a reader is also accessing this register, then this reader might return the

value in the register either before or after the value has been written by the writer. This is a particular problem resulting from the concurrent execution of the read and write operations.

2.5 Distributed Consensus Algorithms and Protocols

Consensus is used by processors to agree on a common value out of values that these processes propose initially. In distributed computing, one of the most fundamental issues is to ensure that multiple processes reach consensus on a common value. In this section, several categories of distributed consensus algorithms are presented and summarized. Here, consensus algorithms are first classified according to failure assumptions. After that, several important protocols are discussed. Based on their complexity, we classify them into two groups: basic and advanced consensus protocols.

2.5.1 *Consensus Algorithms*

A consensus algorithm usually involves two kinds of events, *propose* and *decide*. Each process proposes its initial value for consensus and broadcasts this value as a request to the other processes. All processes then have to decide on the same value among all the proposed values with a decide event. Consensus algorithms can be classified based on the failure assumptions which affect the design of these algorithms. The failure assumptions discussed here include *fail-stop*, *fail-noisy*, *fail-recovery*, *fail-silent*, and *Byzantine* consensus algorithms [25].

Fail-stop consensus algorithms are designed based on the crash-stop process abstraction in the fail-stop model assuming that processes can fail by crashing without recovery. The failure detector abstraction is used so that the crashes can be reliably detected by all the other processes. Also, the broadcast communication abstraction is used for processes to exchange their proposed values and eventually reach an agreement. The broadcast links are assumed to be perfect. The representatives of fail-stop algorithms include Flooding Consensus, Hierarchical Consensus, Flooding Uniform Consensus, and Hierarchical Uniform Consensus [25].

Fail-noisy consensus algorithms assume that processes may fail by crashing as in the fail-noisy model, and that the crashes can be detected by the failure detector abstraction. However, the failure detector used for these consensus algorithms is only eventually perfect and might make mistakes. These algorithms also rely on a majority of correct processes. One representative of these consensus algorithms in the fail-noisy model is known as the Leader-Driven Consensus algorithm [25]. It is based on the leader detector abstraction implemented with the eventually perfect failure detector. This algorithm also provides uniform consensus and runs through a sequence of *epochs* identified using increasing timestamps. For each epoch, a leader process has the task to propose the value for consensus among the processes. The leader process succeeds in reaching consensus if it is correct for its current epoch and no further epoch has started. Otherwise, if a next epoch started, then the algorithm would abort the currently running epoch consensus, obtains its state, and invoke the next epoch consensus with that state.

For addressing consensus with crash-recovery process abstractions of the fail-recovery model, processes may often crash and then recover arbitrarily. That is, in *fail-recovery consensus* algorithms, processes can crash and later recover and still participate. Therefore, they can be considered to be correct if they eventually stop crashing. The algorithms to handle consensus in this fail-recovery model can be implemented by the *logged uniform consensus* abstraction [25] together with the Leader-Driven Consensus algorithm. One representative of these algorithms is known as *Logged Leader-Driven Consensus* [25]. It adapts the Leader-Driven Consensus for the fail-recovery model by logging the current epoch timestamps and leader process pair with the decision value into stable storage such that they can be restored from stable storage after recovery from crashes.

In the *fail-silent consensus* algorithms of the fail-silent model, process crashes can never be reliably detected. Therefore, randomization is considered for consensus without resorting to the failure detector, and the algorithms apply the randomized consensus abstraction. Representatives of these types of consensus algorithms include *Randomized Binary Consensus* [25] which only decides on one bit, and the *Randomized Consensus with Large Domain* [25] which decide on arbitrary large values. Both algorithms need a majority of correct processes to make progress and a *common coin* abstraction [25] for terminating and reaching agreement in a domain of either one bit or arbitrary values.

Consensus with Byzantine process abstractions must allow all processes to reach a common decision despite the presence of faulty processes. However, there are two variants of validity for the *Byzantine consensus* algorithms: weak and strong. The weak variant requires that all processes are correct and propose the same value, and the algorithm only decides the proposed value. For this case, an arbitrary value may be decided by the algorithm if some processes are faulty. On the other hand, the strong variant of the validity of Byzantine consensus tolerates arbitrary-fault processes. If not all of the processes propose the same value, a default value is decided. Byzantine consensus in the fail-noisy-arbitrary model can be implemented with an adapted Leader-Driven Consensus algorithm that works with Byzantine processes. This is known as the *Byzantine Leader-Driven Consensus* algorithm [25]. Alternatively, Byzantine consensus can also be implemented by considering randomization in the fail-arbitrary model. Such Byzantine consensus algorithms are known as the Byzantine randomized consensus. One representative is the extension of Randomized Binary Consensus [25].

2.5.2 Basic Consensus Protocols

In Section 1.5, we have discussed the 2PC protocol, which is a basic consensus protocol. The participants of the protocol achieve consensus if they all agree and accept the proposed value sent by the coordinator and send *YES* votes back to the coordinator.

One of the basic problems with the 2PC protocol is that once the decision *COMMIT* made by the coordinator has reached the participants, the participants follow the decision without checking if other participants got the decision or not. Therefore, the system has no way to confirm if any participant crashed and did not get the decision *COMMIT* message, and the protocol cannot abort. To avoid this problem, one additional communication phase is given for 2PC, providing a *Three-phase Commit (3PC)* protocol [52, 119]. This extra phase is obtained by dividing the second phase

of 2PC into two subphases. For the first subphase, the coordinator sends a *prepare-to-commit* message to all participants after it receives votes in the first phase. Participants who received this message get into the state where they are ready to commit but without doing any unrecoverable operations so that the protocol can be recovered if any participants crashed. Each participant then sends an *acknowledgment* message back to the coordinator indicating that it has received the *prepare-to-commit* message. After that, the last communication phase of 3PC is the same as the second phase of 2PC. Although 3PC improves and fixes the main issue that 2PC has, it still has a problem if the network gets partitioned. The reason is that after the network gets remerged from both partitions, inconsistency may occur due to the different conclusions of the two partitions.

2.5.3 *Advanced Consensus Protocols*

The *Phase King* protocol [17] solves consensus in a synchronous setting with a message passing model. Given n processes with $n > 4f$, the protocol solves consensus with up to f failures. This protocol runs in $f + 1$ phases, each consisting of two rounds. In the first round of each phase, each process broadcasts its preferred value to all other processes for consensus and waits for the values broadcast by others to determine which value is the majority value and its count. In the second round of the phase, a process becomes *king* to other processes if the identity of this process matches the current phase number. Then, this king broadcasts the majority value obtained in the first round. Each process then updates its preferred value to the majority value it observed from the first round if the count of that majority value is greater than $n/2 + f$. Otherwise, the process uses the king's majority value received from the second round as its preferred value.

The *Paxos Consensus* protocol [81] is a family of fault-tolerant consensus protocols used to construct distributed services by allowing a group of server replicas to reach an agreement on one common value among potentially many input values. A whole family of Paxos-based protocols has been developed which focus on different attributes such as latency and throughput. Paxos is often designed with three agent roles: proposers that propose values (one of the proposers is elected as a leader), acceptors that accept a value among those proposed, and learners that learn the chosen value. Each Paxos replica may take on multiple agent roles, e. g., a typical configuration is that all replicas play all agent roles. Paxos can make progress with up to f crash failures, given $n = 2f + 1$ replicas. Paxos has formed the foundation for many production systems, such as Google's Chubby [24] and Spanner [36], and Amazon Web Services [102]. Paxos, however, is also known for being difficult to understand and implement correctly [97].

Raft [104] is a consensus protocol for managing a replicated log. It is developed as an alternative to Paxos since the complexity of Paxos makes it not easy to understand and hard to implement correctly. For the sake of understandability, Raft uses a different structure from Paxos but produces an equivalent result. It separates the consensus problem into several key subproblem elements, including leader election, log replication, safety, and membership changes. For the leader election of Raft, a leader server is elected first from the cluster. This leader has full responsibility for managing log replication on follower servers of the cluster and leads the cluster until

it fails or disconnects in which case a new leader is elected. After electing the leader, this new leader accepts client requests and appends each request into its log. Each request is then forwarded to follower servers of the cluster such that they also update their logs in the same way as the leader does. The leader server uses retransmission to ensure that all of its followers eventually store forwarded requests. Once the leader obtains confirmation from the majority of its followers stating that their logs have been replicated, the leader commits and executes requests in its log. After the followers learn that each request is committed from the leader, they also apply it, which ensures consistency of the logs among all the servers of the cluster.

Proof of Work (POW) [66] is one of the popular consensus algorithms used for blockchains [22] to achieve an agreement on adding new data into distributed servers of the decentralized blockchain network. A blockchain consists of several blocks of data distributed over the decentralized network. Each server in the network has the same copy of the blockchain and all the servers follow the same consensus rules to validate and generate a new block. Therefore, each change in a single blockchain is verified and adopted by other blockchains in the network. The POW achieve consensus for the blockchain by making all the servers in the network solve complex cryptographic puzzles when adding a new block. The server that accomplished this first adds the new block to its blockchain. Then, other servers update their blockchain according to this change.

2.6 Safety and Liveness Properties

When we devise software systems, especially distributed systems and protocols, the executions of the system need to satisfy certain behavioral properties. These properties fall into two main classes: *safety* and *liveness* properties [7, 79]. Distributed services in most cases must satisfy both liveness and safety properties. A safety property is a property that can be violated at some time t, and never be satisfied again after that time. According to safety properties, the distributed system and protocol should not do anything wrong. A liveness property is a property such that, for any time t, the property can be satisfied at some time $t' \geqslant t$. Safety properties ensure that nothing bad happens while liveness properties ensure that eventually, something good happens.

CHAPTER 3

MODEL-BASED TESTING FOR FAULT-TOLERANT DISTRIBUTED SYSTEMS AND PROTOCOLS

Data replication and distributed consensus are central mechanisms for the engineering of fault-tolerant distributed systems protocols, and is widely used in the realization of cloud computing services. Implementing test suites for distributed software systems protocols is a complex and time-consuming task due to the number of test cases that need to be considered in order to obtain high coverage.

This chapter summarizes the articles [132, 133] included in Part II of this thesis. Article [132] explores the use of Coloured Petri Nets (CPNs) for model-based testing of a quorum-based fault-tolerant distributed storage service. We propose the QuoMBT model-based testing framework to test a distributed storage service implemented in the Go programming language based on the Gorums framework [87]. Article [133] applies our proposed model-based testing approach and CPNs to perform model-based testing a more complex case, an implementation of the Paxos distributed consensus protocol. To evaluate our model-based testing approach, we have implemented the Paxos protocol in the Go programming language using the quorum abstractions provided by the Gorums framework. We show how the formal CPN models we constructed can be used to automatically generate a test suite for both the distributed storage service and the Paxos distributed consensus protocol. The testing results show that with the aid of our QuoMBT framework and model-based testing approach, we can obtain high statement coverage for both the distributed storage service and the Paxos protocol.

We first introduce the two systems implemented with the Gorums framework: the distributed storage service and the Paxos distributed consensus protocol. We use the implementations of these two systems together with the Gorums library as the SUTs. After that, we discuss our proposed model-based testing approach and the QuoMBT testing framework. We then present the CPN models constructed for the distributed storage service and the Paxos consensus protocol, and outline how to perform test case generation from the constructed CPN models. At the end of this chapter, we summarize our conclusion and discuss related work.

3.1 Gorums and Distributed Storage Service

We have implemented a distributed storage service, with a single-writer, multi-reader register. This storage service has replicated servers for fault-tolerance. To test this storage implementation, we have designed the CPN model to be discussed in Section 3.4.1

to generate test cases. Both the implementation of the distributed storage and the designed CPN model have been presented in the article [132].

Given the gRPC library, Gorums requires that the server implements the methods specified in the service interface. Therefore, we have implemented two server-side methods: Read() and Write() quorum calls for the distributed storage service, and they can then be invoked as quorum calls from clients, to read/write the state of the storage. In our implementation, only a single write quorum call is allowed to be invoked, but any number of read quorum calls can be invoked by the clients to read the state of the storage. Also, a read may be interleaved with one or more writes generated by the client.

Each replicated server of the storage service maintains a timestamp that is incremented for each new Write() quorum call. The Read() quorum call will always return the value in the storage associated with the highest timestamp as the reply to clients to ensure that the correct value in the storage is picked. Therefore, to implement the reader with Gorums, a user-defined ReadQF quorum function for the Read() quorum call needs to be implemented as shown in Algorithm 1. This ReadQF quorum function collects a set of replies from servers into a single reply that can then be returned from the quorum call; this single reply is determined by the highest timestamp of replies. ReadQSize (the quorum size, defined with an object *qs* of type QuorumSpec) is used to determine if sufficient replies have been received to return the server reply with the highest timestamp.

Algorithm 1 Read quorum function

```
1: func (qs QuorumSpec) ReadQF(replies []ReadReply)
2:     if len(replies) < qs.ReadQSize then                    ▷ read quorum size
3:         return nil, false                                  ▷ no quorum yet, await more replies
4:     highest := ⊥                                           ▷ reply with highest timestamp seen
5:     for r := range replies do
6:         if r.Timestamp ⩾ highest.Timestamp then
7:             highest := r
8:     return highest, true                                   ▷ found quorum
```

3.2 Gorums and Single-decree Paxos

In article [133], we have implemented a single-decree Paxos protocol with a client application using Gorums as our SUT. The system consists of replicas that run the Paxos protocol, handling client requests as input and aiming at reaching consensus on a single output response. Its implementation corresponds to the CPN model for Paxos to be introduced in Section 3.4.2.

In our implementation, each of the Paxos replicas must implement the interface shown in Listing 3.1. This `SinglePaxosServer` interface is generated by the Gorums code generator, based on a set of RPC service methods defined using `protobuf` [51]. The methods `Prepare()`, `Accept()` and `Commit()` in this interface represent Paxos quorum calls that can be invoked by the replicas in order to access and update each other's Paxos state. Also, the `ClientHandle()` method is a quorum call for clients to communicate their proposed value to the Paxos replicas and receive the decided value. The `Ping()`

method is a regular RPC call used by the failure detector to determine if a Paxos replica has failed.

```
type SinglePaxosServer interface {
    Prepare(context.Context, *PrepareMsg) (*PromiseMsg, error)
    Accept(context.Context, *AcceptMsg) (*LearnMsg, error)
    Commit(context.Context, *LearnMsg) (*Empty, error)
    ClientHandle(context.Context, *Value) (*Response, error)
    Ping(context.Context, *Heartbeat) (*Heartbeat, error)
}
```

Listing 3.1: The `SinglePaxosServer` interface that Paxos replicas must implement.

Listing 3.2 shows the main control flow of the single-decree Paxos protocol, where we have omitted error handling and context initialization to shorten the presentation. The first phase of the `Proposer` is to invoke the `Prepare()` quorum call (on Line 3) to send a ⟨Prepare⟩ message to the `Acceptors` which then return ⟨Promise⟩ messages to the `Proposer`. Once a quorum of promises has been obtained, the `Prepare()` quorum call returns with a single combined ⟨Promise⟩ message. Then, the `Proposer` checks the ⟨Promise⟩ message to find if any of the `Acceptors` have voted in a previous round (vrnd). The second phase of the `Proposer` starts by invoking the `Accept()` quorum

```
1  func (p *Proposer) runPaxosPhases() error {
2      preMsg := &PrepareMsg{Rnd: crnd}
3      prmMsg, err := p.config.Prepare(ctx, preMsg)
4      if prmMsg.GetVrnd() != Ignore {
5          p.cval = prmMsg.GetVval()
6      }
7      accMsg := &AcceptMsg{Rnd: crnd, Val: p.cval}
8      lrnMsg, err := p.config.Accept(ctx, accMsg)
9      ackMsg, err := p.config.Commit(ctx, lrnMsg)
10     return nil
11 }
```

Listing 3.2: `Proposer`'s code for Paxos phases (without error handling).

call (on Line 8), asking the `Acceptors` to choose the value included in the ⟨Accept⟩ message. The `Acceptors` respond back with a ⟨Learn⟩ message. For the last phase (on Line 9), the `Proposer` invokes the `Commit()` quorum call to propagate the decision to the `Learners`, which concludes the protocol.

Gorums adds a quorum function signature to an interface called `QuorumSpec` for each quorum call, as shown in Listing 3.3. This interface must be implemented by the protocol developer. One of the benefits of using Gorums's quorum functions is that they are amenable to unit testing (see Section 3.3). The replies from quorum calls are handled by their corresponding quorum functions. As an example, Listing 3.4 shows the implementation of the `PrepareQF` quorum function, which is called by the Gorums runtime with the set of replies that have been received so far. Also, the `PrepareQF` quorum function is called once for each reply, and in the first part (Lines 6-8), it checks if sufficient replies have been returned from the quorum call. If not, then it returns `false`

```
type QuorumSpec interface {
    PrepareQF(replies []*PromiseMsg) (*PromiseMsg, bool)
    AcceptQF(replies []*LearnMsg) (*LearnMsg, bool)
    CommitQF(replies []*Empty) (*Empty, bool)
    ClientHandleQF(replies []*Response) (*Response, bool)
}
```

Listing 3.3: The `QuorumSpec` interface must be implemented to process replies.

```
1  type PaxosQSpec struct {
2      quorum int
3  }
4
5  func (q PaxosQSpec) PrepareQF(replies []*PromiseMsg) (*PromiseMsg,bool) {
6      if len(replies) < q.quorum {
7          return nil, false
8      }
9      reply := &PromiseMsg{Rnd: replies[0].GetRnd()}
10     for _, r := range replies {
11         if r.GetVrnd() >= reply.GetVrnd() {
12             reply.Vrnd = r.GetVrnd()
13             reply.Vval = r.GetVval()
14         }
15     }
16     return reply, true
17 }
```

Listing 3.4: The `PrepareQF` processes ⟨Promise⟩ replies from replicas.

to signal to Gorums that we must wait for more replies. Otherwise, if sufficient replies have been received, a combined ⟨Promise⟩ message is constructed by examining all the replies, and picking the value, *vval*, from the ⟨Promise⟩ message with the highest voted round (*vrnd*). If such a value is found in the replies, this means that the `Proposer` is constrained and must continue to use this value in the remainder of the protocol. Otherwise, the `Proposer` is unconstrained, and can pick its own client value. Similar constructs are used for all the methods in the `QuorumSpec` interface, and these methods are implemented on the `PaxosQSpec` type, which holds information about the quorum size (Line 2).

3.3 Model-based Testing Framework and Approach

We have developed a model-based testing framework called QuoMBT to perform model-based testing of quorum-based distributed systems for protocols implemented using the Gorums framework. Fig. 3.1 gives an overview of the QuoMBT testing framework comprised of CPN Tools and a test adapter. CPN Tools is used for creating a testing model by modeling the SUT, and then generating test cases and oracles via the MBT/CPN library [96] (to be presented in Chapter 4). The generated test cases and oracles are written into XML files. The test adapter consists of a reader and a

tester which can be developed either via manual implementation or by automated code generation. The reader of the test adapter then reads the generated test cases and feeds them into the SUT. Each test case is executed by the tester (included in the test adapter) with the provided test values as inputs. This tester also compares the test oracle's output against the output of each test case in order to determine whether the executed test fails or succeeds.

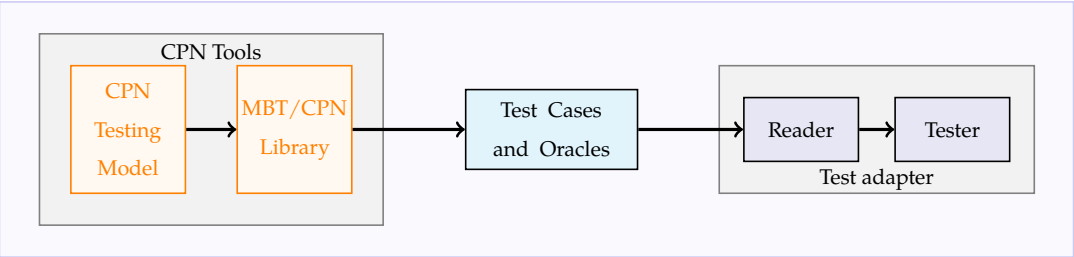


Fig. 3.1: QuoMBT testing framework.

The QuoMBT test framework has been applied to test our Gorums-based implementations of a distributed storage service and a Paxos consensus protocol. Fig. 3.2 illustrates the model-based testing approach with the QuoMBT testing framework when performing model-based testing of quorum-based systems implemented using the Gorums framework.

Our test approach involves four main steps: (a) apply CPN Tools to construct a test model of the SUT; (b) perform test case generation from a CPN testing model to obtain test cases with oracles represented in an XML format; (c) develop a test adapter to execute the generated test cases on the SUT, and compare the test results against generated oracles; (d) develop a test script to start the test adapter and collect testing results. A central part of our test approach for the test case execution is the development of a test adapter which can execute the unit and system test cases generated from CPN Tools using our MBT/CPN library.

For testing the distributed storage service with a single-writer, multi-reader register in the article [132], the test adapter has been implemented in the Go programming language to read XML files containing test cases generated from the CPN model. The execution of tests has been performed by the Go-based tester in the test adapter for executing read and write quorum functions (for unit tests) and quorum calls (for system level tests), with a set of running servers that start first and a client that then invokes quorum calls. The testing results have been captured by the tester and compared against the test oracle. The non-trivial part of the system level tests is the concurrent and sequential executions of read and write quorum calls, which is subject to non-determinism due to concurrent execution of read and write quorum calls. This non-determinism further leads to complexity in obtaining test oracles for system level tests. To cope with this challenge, a run-time monitor was implemented in the test adapter and used to monitor the global correctness of the distributed storage and to obtain valid test oracles.

To perform model-based testing of the implementation of the Paxos consensus protocol [133], the test adapter was also implemented in Go. Here, we distinguish between unit and system tests for the SUT. The unit tests are used to test the central protocol logic used to implement the Paxos consensus protocol. The system tests are

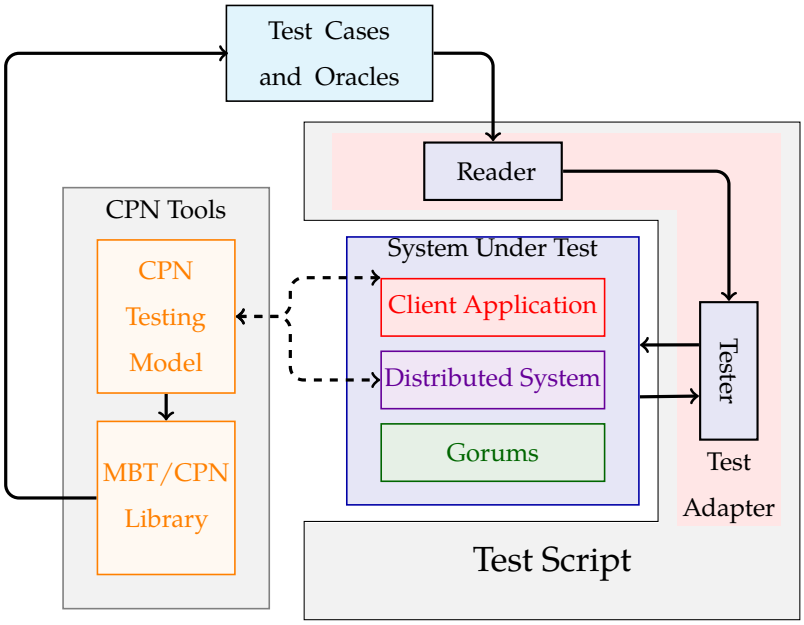


Fig. 3.2: Model-based testing approach with the QuoMBT testing framework.

used to test the complete Paxos implementation and Gorums library with clients. The test adapter can use a reader to read test cases with oracles in the XML format generated from the CPN test model, and use a tester to execute the SUT. Finally, it compares testing results against oracles. As an example, the test adapter can execute two clients concurrently to send their requests to the Paxos replicas. After the Paxos replicas reach consensus, a response value is sent back to the clients. The tester checks whether the response for each client belongs to the expected responses (oracles) and whether the responses are the same for all clients, i. e., consensus is reached.

For both the distributed storage service and the Paxos consensus protocol, our testing approach has been evaluated by measuring statement coverage for both unit and system tests in common successful execution scenarios and scenarios involving server/replica failures. Such measurement is achieved by using a tool provided via the Go testing infrastructure. We have injected programming errors in order to demonstrate if our MBT approach can detect injected errors and investigate to which extent our model-based test cases can detect programming errors. We have also created scenarios which have one or more server failures in order to test fault tolerance of the distributed storage service and the Paxos protocol.

3.4 CPN Testing Models

In this section, the design of CPN models for testing fault-tolerant distributed systems and protocols will be discussed. We first describe the CPN model developed for the distributed storage service [132], and then discuss the CPN model for a Paxos consensus protocol [133]. The Gorums framework has been used to implement both systems, with the aid of two core abstractions: the quorum call and quorum function. Therefore, quorum calls and quorum functions have been considered as main features to be modeled when constructing the CPN models of these two systems and to be tested when performing model-based testing.

With the CPN models, we have generated test cases to perform model-based testing of implementations of these two systems. For the test case generation, we rely on the MBT/CPN library [96], which we have developed to be used with CPN Tools. The details of the MBT/CPN library will be discussed in Chapter 4. The MBT/CPN library is based on extracting test cases from execution sequences of the CPN model by partially observing occurring events. MBT/CPN supports both state space and simulation-based test case generation. State space-based test case generation works for finite-state models and is based on computing all reachable state and state changes of the CPN model. Simulation-based test case generation is based on running a set of model executions and extracting test cases from the corresponding set of execution traces.

3.4.1 CPN Testing Model for a Distributed Storage Service

In article [132], a CPN model has been developed to generate test cases for the Gorums framework and a distributed storage service implementation. The entire system consisting of a set of clients and servers is modeled using CPN Tools. Below we outline the key features of the CPN model. We also use the CPN model to explain the basic operation of the distributed storage service.

Fig. 3.3 shows the top-most module of the CPN model for the distributed storage service, consisting of clients and servers modeled by the substitution transitions Clients and Servers (drawn as rectangles with a double border), respectively. The message channels for communication between the clients and the servers are modeled by the places ClientToServer and ServerToClient. The number of clients and servers are parameters that can be configured without making changes to the net-structure.

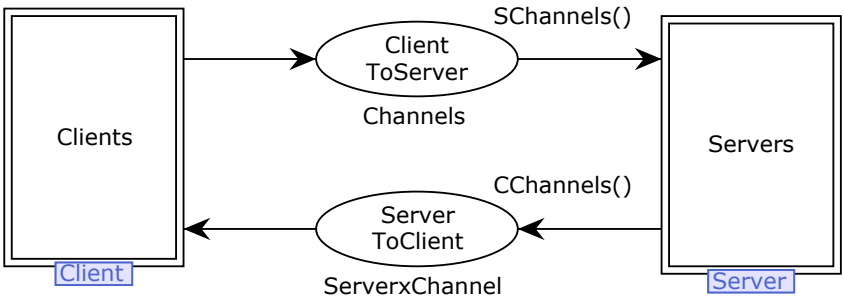


Fig. 3.3: Top-level module of the CPN model for the distributed storage service.

Fig. 3.4 shows the client submodule of the Clients substitution transition in Fig. 3.3. The behavior of applications running on the clients consists of read and write quorum calls provided by the distributed storage. The read quorum call is used by clients to read data from the storage; the write quorum call is used by clients to write data into the storage. The details of quorum calls are modeled by the substitution transitions Read and Write. Additionally, the substitution transition QuorumCalls has submodules that serve as test driver modules used to generate test cases for the implementation of the distributed storage and the Gorums framework. The invocation of quorum calls is done by placing tokens on the Read and Write places. The port places ServerToClient and ClientToServer are linked to the identically named socket places in Fig. 3.3.

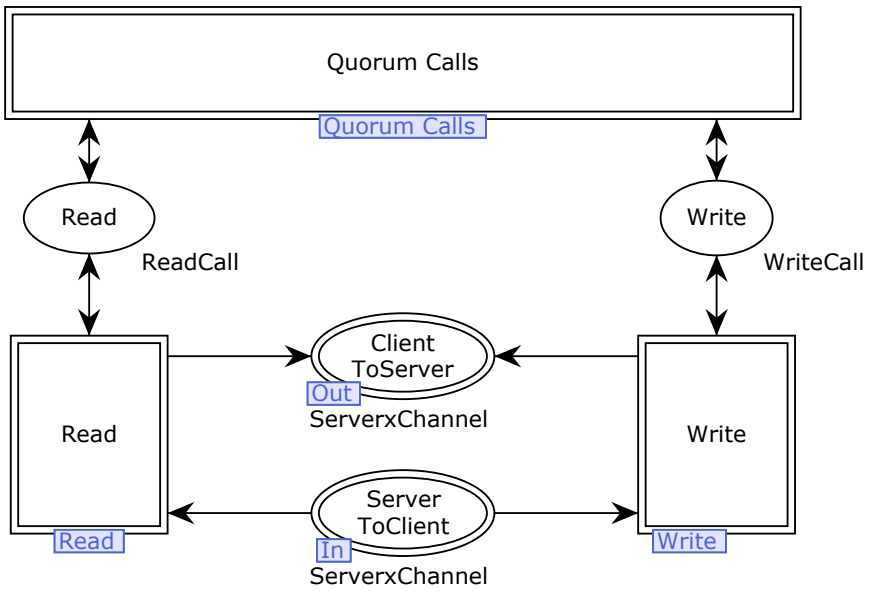


Fig. 3.4: The Clients module for the distributed storage service.

Fig. 3.5 shows the submodule of the Read substitution transition which provides an abstract implementation of the read quorum call. The main purpose of the Read module is to generate test cases for the read quorum function. The execution of a read quorum call starts by sending a read request to each of the servers, which is modeled by the transition SendReadReq and the expression on the arc to place ClientToServer. The place ReadReplies is used to collect the replies received from the servers. After sending a read request to each of the servers, the read call enters a WaitingReply state and waits for replies coming back from the servers. When a read reply comes back represented as a token on place ServerToClient, then transition ApplyReadQF will be enabled, and the read quorum function is then invoked, as represented by the arc expressions to WaitingReply and Read. With sufficient replies received, a read result is returned to the Read.

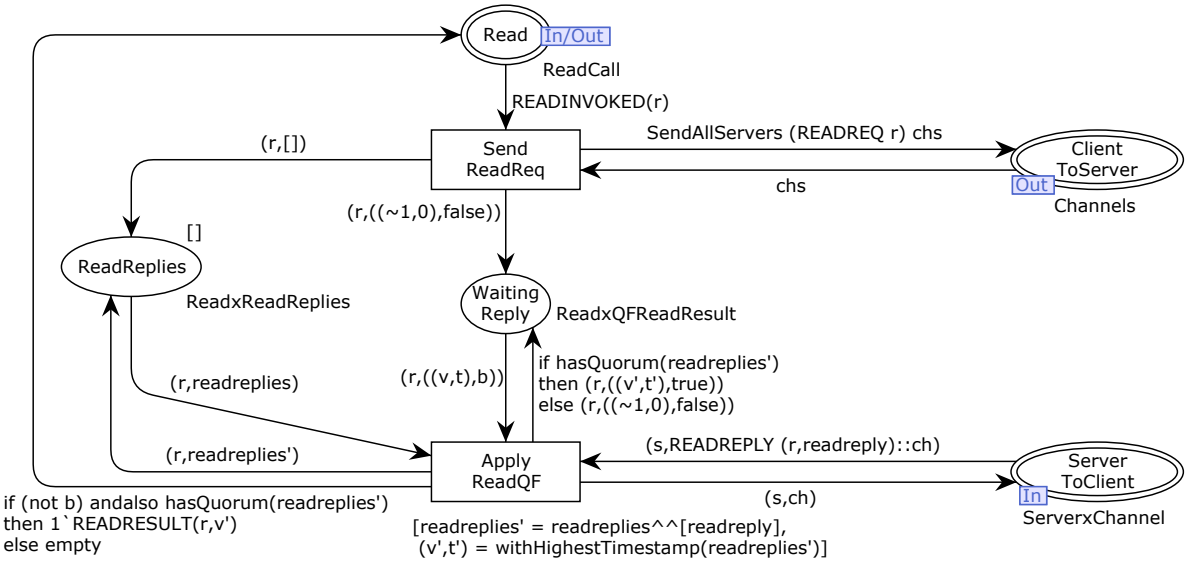


Fig. 3.5: The Read module for the distributed storage service.

Fig. 3.6 shows the server submodule of the Servers substitution transition in Fig. 3.3. The replicated state of each server is modeled by the place State. The two substitution transitions are used for modeling the handling of write requests and read requests on the server side. As an example, the processing of a write request from a client is modeled by the submodule of the HandleWriteRequest substitution transition shown in Fig. 3.7. The place ClientToServer (upper right) receives the incoming write request presented as a value in the list-token, with a value `v'` to be written into the distributed storage and a timestamp `t'`. The new value is stored on the server only if the timestamp `t'` of the incoming write request is larger than the timestamp `t` for the currently stored value `v`. If so, then the new value `v'` is stored on the server, and a write acknowledgment is sent back in a write reply to the client. Otherwise, the stored value remains unchanged and a negative write acknowledgment is sent to the client in the write reply. The handling of read requests is modeled in a similar manner, except that no comparison is needed, and the server simply returns the currently stored value together with its timestamp.

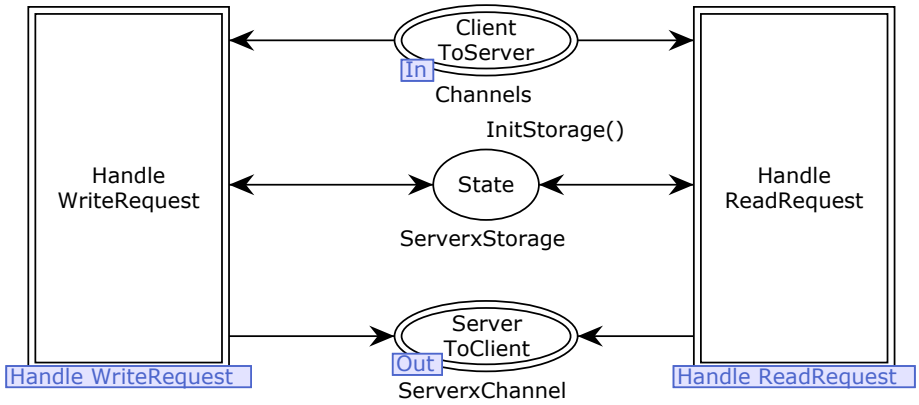


Fig. 3.6: The Server module for the distributed storage service.

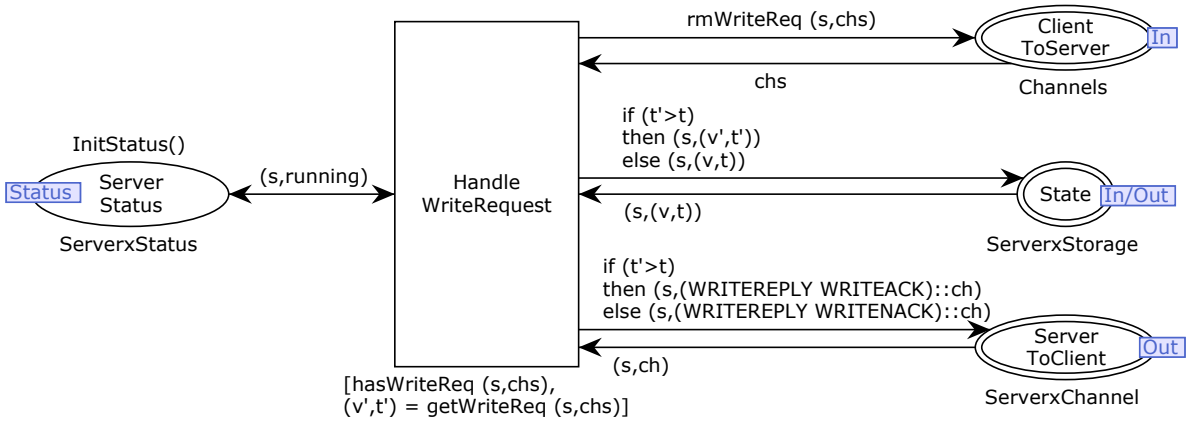


Fig. 3.7: The HandleWriteRequest module for the distributed storage service.

The generation of test cases for the distributed storage system and Gorums is based on the analysis of executions of the CPN model. Test cases can be generated for both the read and write quorum functions as unit tests and the quorum calls as system level tests consisting of concurrent and interleaved invocations of read and write quorum

calls. System level tests can test both the implementation of the quorum calls and the Gorums framework implementation itself. In addition to the test cases, we also generate a *test oracle* for each test case to determine whether the test passes or not. The state space for the CPN testing model of the distributed storage service is relatively small, and we can obtain all test cases based on state space-based test case generation. These test cases are generated and represented using XML.

Listing 3.5 shows an example of how test cases are represented using XML in order to perform unit test of the ReadQF quorum function of the distributed storage service. The test case for the ReadQF quorum function has two replies, one with value 0 and timestamp 0; the other with value 42 and timestamp 1. With the configuration of three servers, this constitutes a quorum, and the value returned from the quorum function is therefore expected to be 42 with the timestamp of 1.

```
<Test TestName="ReadQFTest">
    <TestCase CaseID="1">
        <TestValues>
            <Content>
                <Value>0</Value>
                <Timestamp>0</Timestamp>
            </Content>
            <Content>
                <Value>42</Value>
                <Timestamp>1</Timestamp>
            </Content>
        </TestValues>
        <ExpectResults>
            <Value>42</Value>
            <Timestamp>1</Timestamp>
        </ExpectResults>
        <ExpectQuorum>true</ExpectQuorum>
    </TestCase>
</Test>
```

Listing 3.5: Example test case generated for read quorum function.

For system level tests involving read and write quorum calls, the generation of test cases and expected results is based on a submodule of the CPN model. In this case, it is performed by the module QuorumCalls which acts as a *test driver* to specify different scenarios of read and write quorum calls of the quorum system. By varying this module, it is possible to generate test cases with expected results for different scenarios of read and write quorum calls.

Fig. 3.8 shows an example of a test driver in which the client executes one read and one write quorum call as modeled by the transition InvokeRDWR. After completion of these two calls, server failures may occur and a new read and a write call is invoked as modeled by the transition InvokeRDWRFailures.

Based on this, test cases can be generated in XML format specifying both the concurrent and sequential execution of read and write calls. Listing 3.6 shows an example where first a read and a write are initiated. After completion of these two calls, a new read call is initiated. We handle concurrent executions by nesting the read and write `Routine` tag as illustrated in Listing 3.6, while non-nested `Routine` tags are considered sequential.

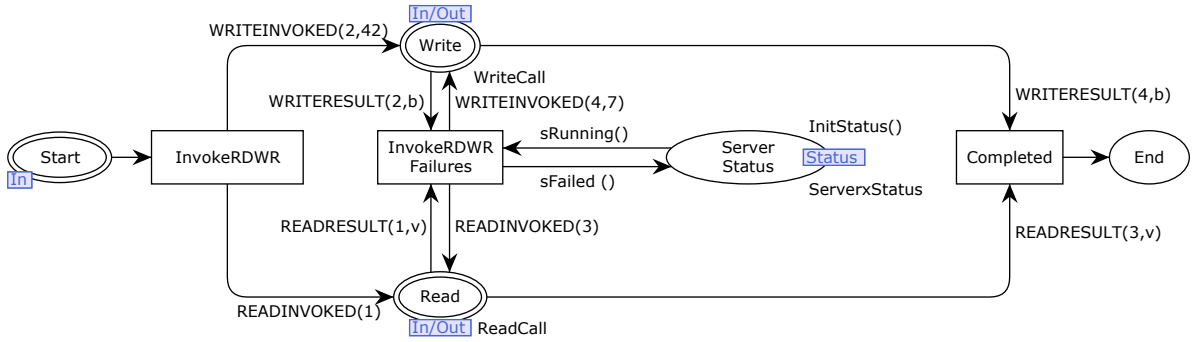


Fig. 3.8: The QuorumCalls module for the distributed storage service.

```
<Test TestName="SystemTest">
 <TestCase CaseID="WRprRDsqRD">
     <Routine RoutineID="A" OperationName="Write">
         <OperationValues>
             <Value>7</Value>
         </OperationValues>
         <Routine RoutineID="B" OperationName="Read">
             <OperationValues>
                 <Value>7</Value>
                 <Value></Value>
             </OperationValues>
         </Routine>
     </Routine>
     <Routine RoutineID="A" OperationName="Read">
         <OperationValues>
             <Value>7</Value>
         </OperationValues>
     </Routine>
 </TestCase>
</Test>
```

Listing 3.6: Example test case generated for the concurrent and sequential execution of read and write calls.

3.4.2 CPN Testing Model for Single-decree Paxos

The single-decree Paxos consensus protocol can be used by a distributed application in which the Paxos replicas need to agree on a single common value among potentially many input values. We assume that one or more clients send the input values to the Paxos replicas, and then receive the decided output value returned from Paxos replicas. The complete CPN model of the single-decree Paxos protocol is discussed in article [133] included in Part II. The CPN model is comprised of 23 hierarchically organized modules. We introduce the key elements of the CPN model below.

Paxos is usually explained with three separate agent roles: *proposers*, *acceptors* and *learners* [82, 97]. Proposers can propose values for consensus; acceptors accept a value among those proposed to reach consensus; learners learn the chosen value. A Paxos replica may play multiple agent roles. As a typical configuration, all replicas play all agent roles. The constructed CPN model reflects this typical configuration. Moreover,

Paxos is safe for any number of crash failures, and given $n = 2f + 1$ acceptors, it can make progress with up to f crash failures.

Fig. 3.9 shows the top-level module of the CPN model consisting of the two substitution transitions Clients and Replicas connected by the two places Request and Response. The behavior of the clients is modeled by the substitution transition Clients with its associated submodule Clients. Clients send request value for consensus to the Paxos replicas, by putting a token on the socket places Request, and wait for the decided response value to be returned as a token on socket places Response. The behavior of the distributed replicas is modeled by the substitution transition Replicas and its associated submodule Replicas, with the aim of executing the Paxos protocol to reach consensus on a value proposed by the clients.

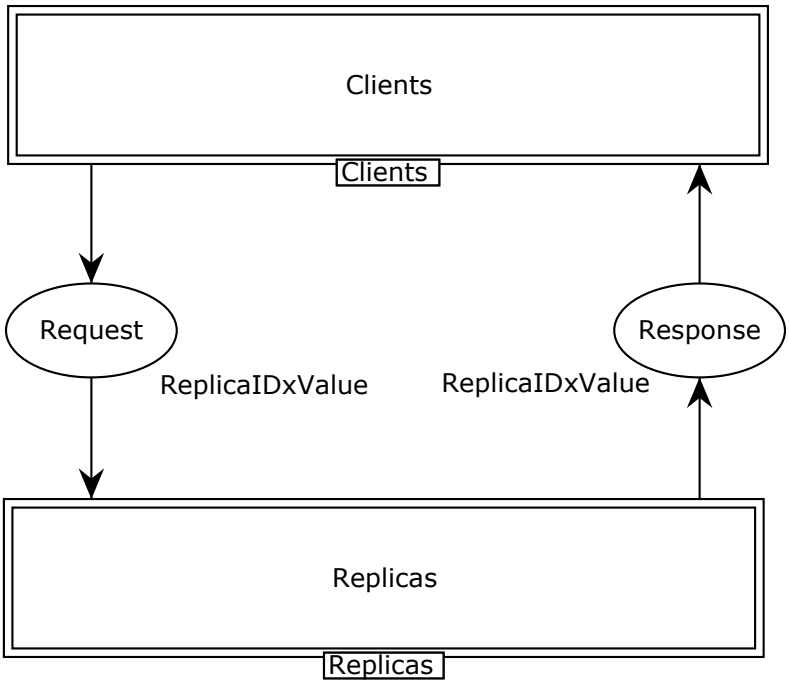


Fig. 3.9: Top-level CPN module for the single-decreee Paxos model.

Based on the definitions of Paxos agent roles, Fig. 3.10 shows the Replicas module (the submodule of the substitution transition Replicas in Fig. 3.9), which consists of three substitution transitions for three Paxos agents connected by socket places to model the communication between the different agents. Each substitution transition then models the detailed behavior of a Paxos agent. The Replicas module is constructed such that any number of replicas can be configured without changing the net-structure. This make it easy to generate test cases for different size of Paxos configurations.

The Paxos protocol operates in *rounds*, and usually each round of the protocol is associated with a single proposer, which is the *leader* for that round and executes three communication phases between replicas:

1. A proposer sends a ⟨Prepare⟩ message to the acceptors and collects at least $f + 1$ ⟨Promise⟩ messages;

2. The proposer then sends ⟨Accept⟩ messages for some value v to the acceptors,

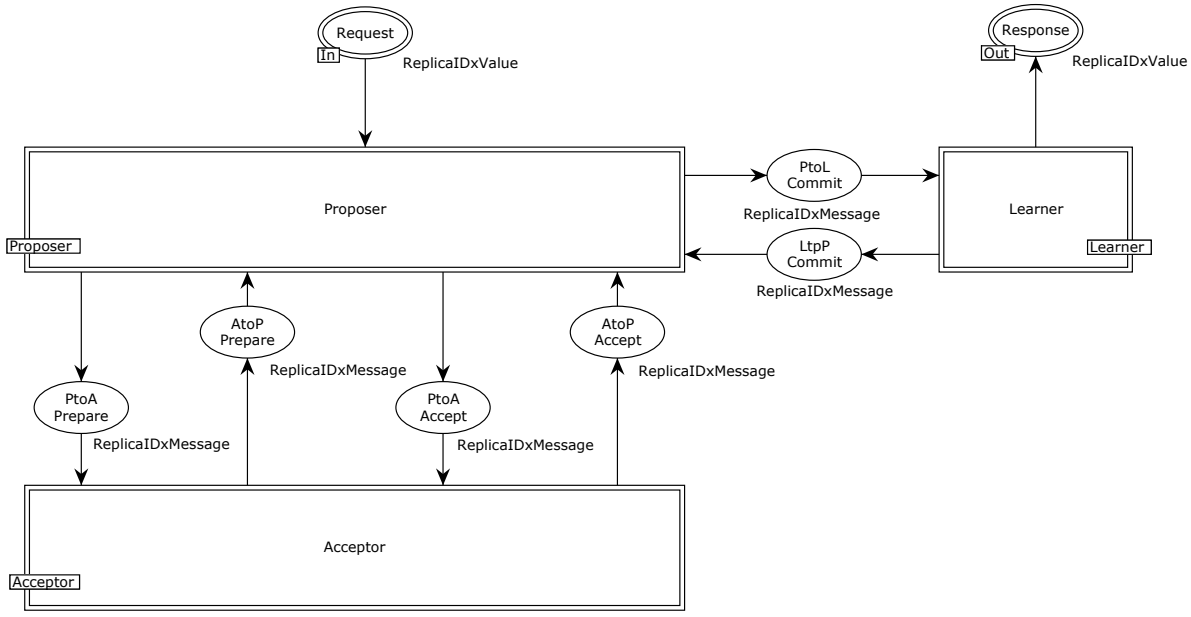


Fig. 3.10: The single-decree Replicas module.

that respond by sending ⟨Learn⟩ messages back to the proposer acknowledging the value v;

3. The proposer sends the decided value in ⟨Commit⟩ messages to learners.

The protocol solves consensus in some round, but due to asynchrony and failures, it may need to run several rounds to reach consensus. That is, if the current proposer (leader) goes mute, another proposer can be elected as a new leader to start a new (higher) round.

Fig. 3.11 shows the submodule of the Proposer substitution transition, which contains three substitution transitions: LeaderDetector, FailureDetector, and ProposerCore. The Proposer of each replica receives the client request sent from the submodule of the Clients substitution transition for consensus; this request is presented as a token on the port place Request. The LeaderDetector substitution transition and its submodule model the behavior of electing a leader among the Proposers. The FailureDetector substitution transition has a submodule that can detect the failure of any of the Proposers. If a failure is detected by the failure detector, the submodule of the LeaderDetector substitution transition will be informed via the submodule of the FailureDetector substitution transition.

The submodule of the ProposerCore substitution transition models the behavior of the initialization of Proposers and the behavior of three Paxos communication phases. Fig. 3.12 illustrates a successful round of three communication phases for single-decree Paxos protocol. The first number in each message is the current round number $\mathtt{rnd} = 1$, and v in each message is the value that the proposer wants the acceptors to choose. The gray boxes labeled v represent the execution of a state machine command derived from the decided value v. While not shown in the figure, each replica has instances of each of the Paxos agents. For each Paxos communication phase, the communication between the different Paxos agents is modeled based on the quorum abstractions provided by

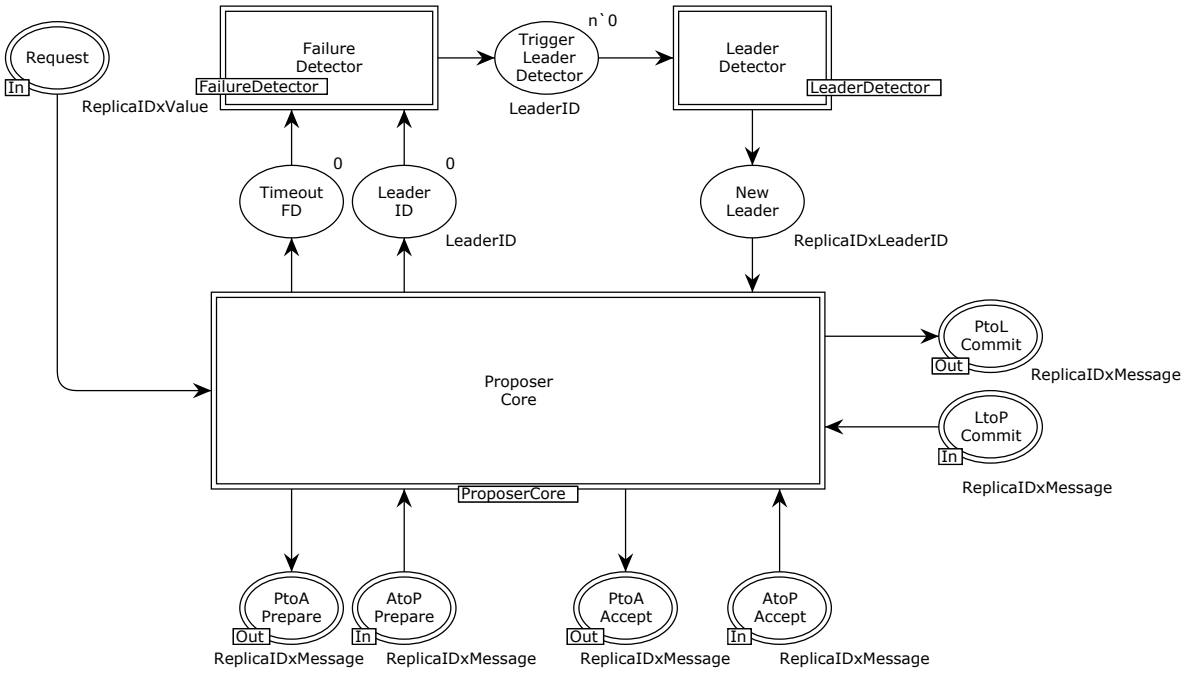


Fig. 3.11: The Proposer module.

the Gorums framework [87]. Specifically, the communication takes the form of quorum calls, one for each of the Paxos communication phases: Prepare, Accept, and Commit.

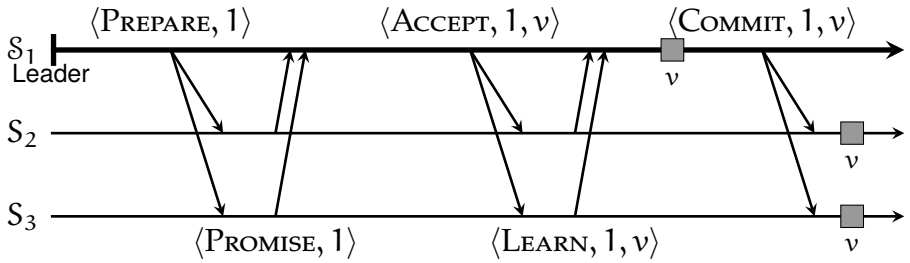


Fig. 3.12: The single-decree Paxos consensus protocol with three phases.

The first communication phase involves two types of messages known as the ⟨Prepare⟩ and ⟨Promise⟩ messages as shown in Fig. 3.12. The leader candidate creates a ⟨Prepare⟩ message with its current round number and invokes a Prepare quorum call to send the ⟨Prepare⟩ message to Acceptors, aiming at proposing itself to be a leader. The Prepare quorum call is modeled by the submodule of the Prepare substitution transition. After the Acceptors receive the ⟨Prepare⟩ message, and if they accepted it, then each Acceptor returns back a ⟨Promise⟩ message to the leader candidate by the Prepare quorum call. This behavior of the Acceptors is modeled by the submodule of the Acceptor substitution transition in Fig. 3.10. When the leader candidate receives enough ⟨Promise⟩ messages to obtain a quorum, then the first phase is finished. The leader candidate now can become a leader to propose the client request to Acceptors for consensus.

In the second phase, two types of messages are involved, known as ⟨Accept⟩ and ⟨Learn⟩ messages. The leader creates an ⟨Accept⟩ message with its current round number, crnd, and the value v obtained from the client request, then invokes the Accept

quorum call, modeled by the submodule of the Accept substitution transition. This quorum call sends the ⟨Accept⟩ message to the Acceptors to let them vote for consensus value v. After the consensus value v is chosen, then the Acceptor will return a ⟨Learn⟩ message to the leader. Once the leader receives a quorum of ⟨Accept⟩ messages from Acceptors, the second phase is done. For the third phase, the leader invokes the Commit quorum call on the Learners to send ⟨Commit⟩ messages. This enables the Learners to learn the chosen consensus value and they can send it to the clients. The behavior of the Learners is modeled by the submodule of the Learner substitution transition in Fig. 3.10.

The submodule of the ProposerCore substitution transition has substitution transitons Prepare, Accept, and Commit to model the Prepare, Accept and Commit quorums calls. Fig. 3.13 gives the Prepare quorum call module of the Prepare substitution transition. This module models the behavior of the quorum call and quorum function abstractions provided by Gorums for sending the ⟨Prepare⟩ messages from a Proposer (leader) to Acceptors when the transition SendPrepareMessages occurs. Then, after such ⟨Prepare⟩ messages have been handled by Acceptors, the ⟨Promise⟩ messages from Acceptors can be processed when the transition ApplyPrepareQF occurs, which models the behavior of the Prepare quorum function. The logic of quorum functions was already discussed in Listing 3.4 in Section 3.2.

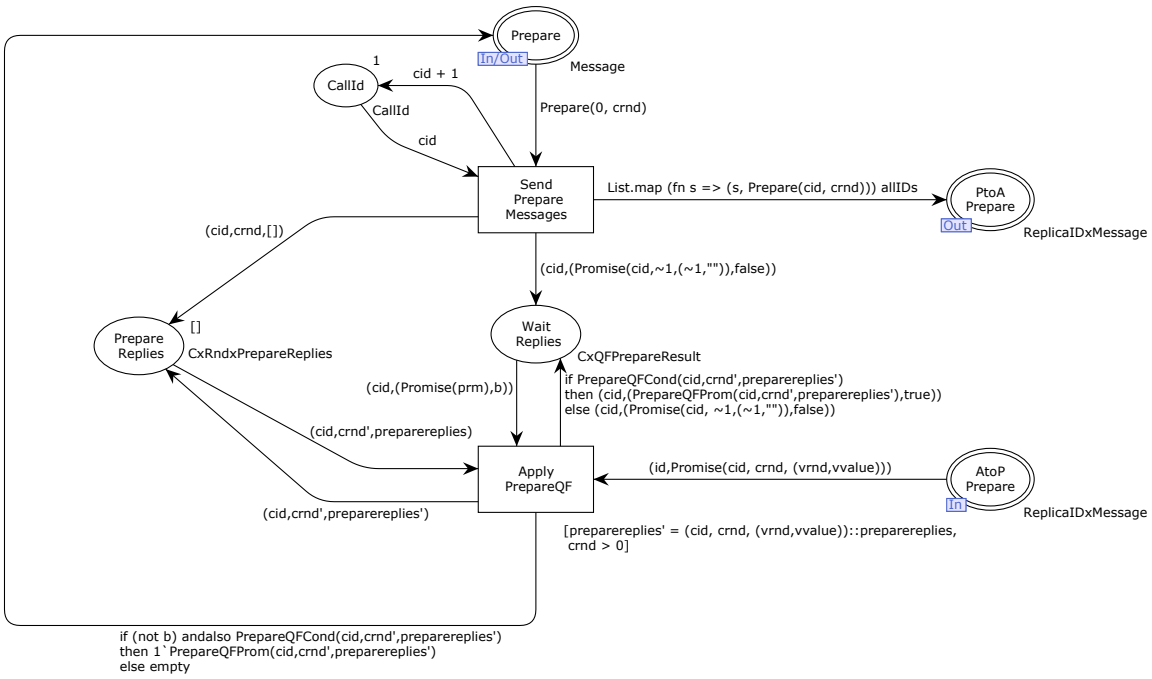


Fig. 3.13: The Prepare quorum call module.

To generate test cases with oracles for the Paxos consensus protocol, we rely on the simulation-based approach, due to the complexity of the Paxos CPN model. Both unit test and system tests are generated for the Paxos protocol. The unit test are concerned with testing of the quorum functions, which forms the core of the Gorums-based implementation for Paxos protocol. The system level tests are concerned with the proposed values, chosen value for consensus, selected leaders, and failure of replicas.

For unit tests, Listing 3.7 shows an excerpt from the XML representation of a test

```
<Test Name="TestPrepareQF">
    <TestCase ID="1">
        <TestValues>
            <PromiseMsg>
                <Rnd>0</Rnd>
                <Vrnd>0</Vrnd>
                <Vval></Vval>
            </PromiseMsg>
            <PromiseMsg>
                <Rnd>0</Rnd>
                <Vrnd>0</Vrnd>
                <Vval></Vval>
            </PromiseMsg>
        </TestValues>
        <TestOracles>
            <Quorum>true</Quorum>
            <PromiseMsg>
                <Rnd>0</Rnd>
                <Vrnd>0</Vrnd>
                <Vval></Vval>
            </PromiseMsg>
        </TestOracles>
    </TestCase>
</Test>
```

Listing 3.7: Example test case generated for the PrepareQF() of the Paxos protocol.

case for `PrepareQF()`, which corresponds to a test case where Paxos is configured with three replicas and the quorum size is two. The test input for the `PrepareQF()` method in the test case is two ⟨Promise⟩ messages with values for the fields Rnd, Vrnd and Vval. The expected output of the `PrepareQF()` is a ⟨Promise⟩ message together with the Quorum boolean *true*, indicating that a quorum was obtained for these input messages.

```
<Test Name="systemtest">
    <TestCase ID="1">
        <TestValues>
            <ClientPropose>M1</ClientPropose>
            <ClientPropose>M2</ClientPropose>
            <P1Failure>1</P1Failure>
        </TestValues>
        <TestOracles>
            <Leader>0</Leader>
            <Leader>1</Leader>
            <Response>M1</Response>
            <Response>M2</Response>
        </TestOracles>
    </TestCase>
</Test>
```

Listing 3.8: Example test case generated for the Paxos system with three replicas.

For system tests, Listing 3.8 shows an example of a test case for the Paxos protocol with three replicas configured, and a failure in the first Paxos phase. The test input

for this example consists of two clients sending requests concurrently to the Paxos replicas. The test oracles include the valid responses from Paxos replicas, and the expected leaders. Leader 0 is the first leader, and after it fails, leader 1 becomes the new leader. It can be used to check whether the correct leaders are chosen, and whether the response returned to each client belongs to the set of legal responses. Furthermore, it also checks whether the responses obtained by all clients are equal, so that we can determine if consensus is reached or not.

3.5 Results and Contributions

The main contribution of our work in articles [132, 133] is a model-based testing approach based on formal modeling which can be used for testing advanced quorum-based distributed systems and protocols. Our approach includes modeling patterns, test case generation algorithms, and a test case execution infrastructure. As case studies for evaluation of our approach, we have considered a distributed storage system and a Paxos implementation. We have shown that our QuoMBT testing framework can perform both unit tests for the quorum logic functions and system level tests for quorum calls, with both successful scenarios and scenarios involving failures and programming errors. In addition to obtaining high code coverage, our generated unit and system tests can detect programming errors in the implementation.

Table 3.1 summarizes our experimental results for the distributed storage system for one of the complex test drivers, designed in [132], involving a concurrent execution of read and write quorum calls followed by another read quorum call ((WR||RD);RD). The results shows that

1. for the successful scenario, we obtain 100 % of code coverage for the quorum functions (unit test), 84.4 % of statement coverage on the quorum calls (system test), and 40.8 % of statement coverage on the Gorums framework as a whole;

2. for the scenario involving failures (e. g., crash of servers) and injected programming errors, we also obtain 100 % of code coverage for the quorum functions with unit tests, and the statement coverage on quorum calls and Gorums as a whole increase to 96.7 % and 52.3 %, respectively.

Table 3.1: Experimental results for distributed storage system

Test Driver	Scenarios	Test Case Execution				
		System			Unit	
		Gorums Library	QCs RD	QCs WR	QFs RD	QFs WR
(WR||RD);RD	Success	40.8	84.4	84.4	100	100
	Failures/ Errors	52.3	96.7	96.7	100	100

As part of analyzing the results of the code coverage, we also discovered a code path not covered in Gorums. So we added an additional test that can cover this particular path, which involves passing *nil* as an argument to either read or write quorum calls.

The test case revealed an error in this code path causing the test client to crash. The error has been reported to the Gorums developers, and a fix has been implemented.

Table 3.2 summarizes the experimental results obtained for the Paxos implementation [133]. The results show the statement coverage for the different subsystems of our Paxos implementation. Note that Unit tests are only applicable for the quorum functions, not for subsystems. Specifically, we obtain 90 % and 85.7 % of code coverage for the Prepare and Accept quorum functions of unit tests, respectively. For the system tests, the statement coverage for three different quorum calls reaches 83.9 %, respectively; for Prepare and Accept quorum functions, the statement coverage are up to 100 %; for the Paxos core implementation, the Proposer module's statement coverage reaches 97.4 %; the Acceptor module's statement coverage is up to 100 %; the statement coverages of the Failure Detector and Leader Detector modules reach 75.0 % and 91.4 %, respectively; the Paxos replica module reaches 91.4 % of the statement coverage; for the Gorums library as a whole, the highest statement coverage reaches 51.8 %. These results indicate that our MBT approach and the QuoMBT testing framework are promising in terms of obtaining a high statement coverage of the system under test via generated test cases.

Table 3.2: Experimental results for Paxos consensus protocol.

Subsystem	Component	System tests	Unit tests
Gorums library		51.8 %	-
Paxos core	Proposer	97.4 %	-
	Acceptor	100.0 %	-
	Failure Detector	75.0 %	-
	Leader Detector	91.4 %	-
	Replica	91.4 %	-
Quorum calls	Prepare	83.9 %	-
	Accept	83.9 %	-
	Commit	83.9 %	-
Quorum functions	Prepare	100.0 %	90.0 %
	Accept	100.0 %	85.7 %

An important attribute of our approach is that the CPN testing models are constructed such that they can serve as a basis for model-based testing of *other* quorum-based distributed systems and protocols implemented with the abstractions of the Gorums framework. In other words, given a distributed system implemented by the Gorums framework, it is only the implementation of the quorum functions that needs to be changed when modeling the behaviors of quorum calls and quorum functions. The state space and simulation-based test case generation approaches are independent of the particular quorum system under test. This benefit can be seen from the Read module in Fig. 3.5 of the CPN model for the distributed storage service and the Prepare quorum call module in Fig. 3.13 for the Paxos consensus protocol. Both modules use a similar modeling pattern for the quorum call and quorum function of the Gorums framework.

3.6 Related Work

Model-based testing is a large research area with approaches and tools that have been developed based on a variety of modeling formalism. These modeling formalism include flowcharts, decision tables, finite-state machines, Petri Nets, state-charts, object-oriented models, and BPMN [69]. In Saifan and Dingel's survey [114], a detailed description is given on how model-based testing is effective in testing different aspects of distributed systems. The survey classifies model-based testing based on different criteria and compares several model-based testing tools for distributed systems based on this classification. The comparison does not identify work that can be applied to systems that rely on a quorum system to achieve fault-tolerance.

Model-based testing has been successfully used (as measured through productivity gain) in Microsoft's Protocol Documentation Quality Assurance Process. Grieskamp et al. [54] used Spec Explorer on protocols, where a so-called model program describes the test case, including how to check an observation against a possibly non-deterministic outcome. The main difference to our work is that their model programs are rule-based, and as such only results in a visual representation as a graph through state space exploration. Our CPN models give developers a better overview as they directly link client- and server interactions, and hence make the behavior of the protocol explicit.

Chubby [26] was one of the first implementations of Paxos that were deployed in a production environment, and thus were extensively tested. The authors highlight that it was unrealistic to prove correct a real system of that size at the time (2007). Thus, they adapted meticulous software engineering practices achieving robustness, and tested their system thoroughly. One of their testing strategies was to test their implementation when suffering a random sequence of network outages, message delays, timeouts, process crashes and recoveries, and schedule interleaving. Given our CPN models and generated tests, we aim to test many of the same attributes but in a more systematic manner.

The ZooKeeper distributed coordination service [64] has been successfully tested by Modbat [11] in a similar setting as ours. To test ZooKeeper, Modbat explores different possible interleaving and non-deterministic outcomes due to scheduling decisions or network communication in the real system which are judged by an oracle essentially implementing a model checking component. Unlike our CPN models, the specifications are not for consumption by other tools such as model checkers, nor is there an interactive component that allows exploring a particular execution of the model. As in our approach, it might require manual effort to connect the engine to the SUT. Our CPN models can be used to test quorums of the distributed systems and protocols, and Modbat has not been used to test the majority quorums of the ZooKeeper service intended to guarantee a consistent view of the system.

Formal verification of distributed protocols copes with protocols on a more abstract level. It aims at finding flaws and inconsistencies primarily in the specification. Such approaches are not necessarily targeting a correct implementation, and only rarely can executable code directly or automatically be derived from the specification. Formal verification for such complex system protocols often suffers from undecidability issues. These issues require careful management of any automation [59], or substantial effort to encode the system in a decidable fragment [106]. We consider our model-based

testing approach for a concrete implementation as orthogonal to approaches that aim to validate and verify the correctness of a protocol in general. Frequently, the final, often manual step of actually programming a proven-as-correct algorithm introduces mistakes, and generated code may also suffer from problems or assumptions about the underlying infrastructure as found, e. g., in Fonseca's analysis of IronFleet [47].

A CPN-based simulation method [62] has been proposed and applied to a quorum-based distributed storage system called Cassandra [9]. The focus of this research work was to find appropriate parameter settings to achieve the best performance, since Cassandra is highly configurable. The authors developed a CPN-based simulator specifically for Cassandra, which allow tuning various system parameters such as cluster size, timeouts and read/write ratios, for their CPN models. In our work, we focus on using the CPN testing models for generating test cases to perform both unit and system tests for the implementations of distributed systems and protocols.

Ponce de León et al. [107] have discussed a testing approach for true concurrency using I/O Petri nets. The authors define a concurrent conformance relation for input-output labelled transitions systems, IOLTS. A test case selection algorithm has been proposed with criteria such as covering all paths of length n, or traversing each basic behavior a certain number of times. Test case selection is also a challenge in our setting with CPN models, so it remains an open question how their unfolding would work in our CPN setting. Watanabe and Kudoh [135] propose two different CPN-based test suite generation methods for concurrent systems. Their methods do not directly address a particular way to derive a CPN testing model for a distributed system, nor do they analyze achieved code coverage.

Zheng et al. [140] describe two algorithms for generating test cases and test sequences from a CPN model of the SUT. This CPN model is first used as input to their APCO algorithm to generate an initial set of test cases. These test cases can then be converted to test sequences using their algorithm. Then, the set of original test cases and test sequences can be exported as XML formatted files. The authors have applied their technique to a radio module in a centralized railway control system. In contrast to our approach, Zheng et al. do not consider testing any failure scenarios of systems and protocols. Also, they do not consider concurrent execution of processes, and their approach has not been used to validate distributed software systems.

CHAPTER 4

A SOFTWARE TOOL FOR TEST CASE GENERATION WITH COLOURED PETRI NETS

Society is heavily dependent on software and software systems, and design- and implementation errors in systems may render them unavailable and return erroneous results to the users. It is therefore important to develop automated techniques and supporting software tools that can be used to support the engineering of correct and stable software systems. This chapter summarizes the MBT/CPN software engineering tool as presented in the article [134] and discusses the automated MBT approach using the developed test execution engine.

The MBT/CPN tool augments CPN Tools with facilities for model-based test case generation, and is based on the user identifying observable events formalized in a so-called test case specification. As we will illustrate on the two-phase commit transaction (2PC) protocol, this entails implementing a detection, observation, and formatting function which is applied during the test case generation. An important feature of the MBT/CPN tool is the uniform support for test case generation based on state spaces and simulation. We show by practical experiments on the 2PC protocol, the distributed storage protocol, and the Paxos consensus protocol that we can obtain a high SUT code coverage, and that our approach can be used to detect implementation errors [132–134].

In this chapter, we first present an overview and the architecture of the MBT/CPN tool, implemented through CPN Tools, to support test case generation from CPN models. We use the 2PC protocol as a running example to present the features of the MBT/CPN tool, and discuss our two approaches implemented in the MBT/CPN tool for test case generation. We then introduce our automated MBT approach with the aid of the MBT/CPN tool for test case generation, and the test execution engine to automatically generate test adapters for test case execution. Finally, we sum up contributions and discuss related work.

4.1 Software Architecture of the MBT/CPN Software Tool

Fig. 4.1 gives an overview of the modules that constitute the MBT/CPN tool and how it is applied in the context of model-based test case generation. The MBT/CPN tool is implemented in the Standard ML programming language on top of the simulator of CPN Tools. The main output of the MBT/CPN tool are XML files containing **Test Cases**. Based on the generated test cases, a **Test Adapter** can employ a **Reader** to read

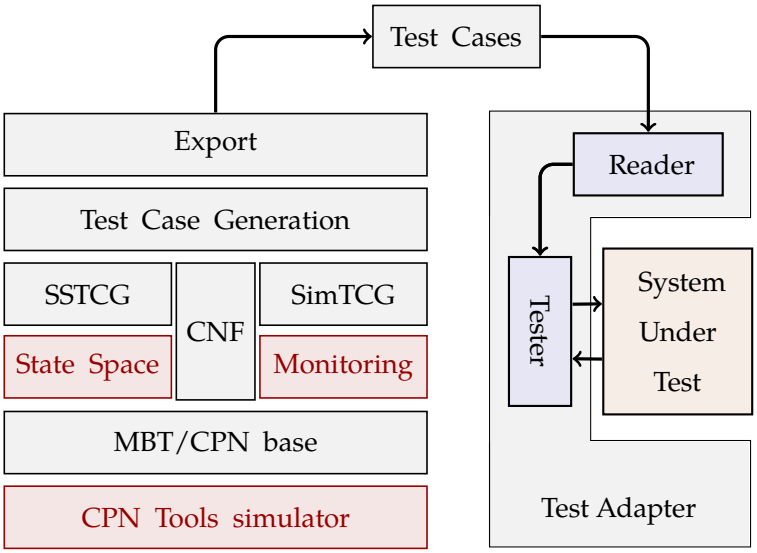


Fig. 4.1: Overview of MBT/CPN modules.

test cases and a **Tester** to execute them against the **System Under Test** (SUT). This **Tester** provides input events to the SUT and compares the observed outputs from the SUT with the expected outputs. One benefit of using XML as output format from the MBT/CPN tool is that any programming language that support XML can use the generated test cases. This gives the flexibility to test systems and software implemented in different programming languages.

To use the MBT/CPN tool, the user must identify the *observable events* that can be derived from the binding elements in the CPN model. A binding element represents a mode of a transition that may be enabled and may occur. A test case is comprised of observable events consisting of input events and expected output events. Input events represent stimuli to the SUT, while expected output events represent test oracles and are used to determine the test outcome during test case execution.

The observable events in test cases are represented by the generic colour set (data type) `TCEvent` defined as:

```
colset TCEvent = union InEvent:TCInEvent + OutEvent:TCOutEvent;
```

This generic colour set is defined by the **MBT/CPN base** module in Fig. 4.1. The user of the tool must give the definition of the colour sets `TCInEvent` and `TCOutEvent` which depend on the observable events of the SUT.

The MBT/CPN tool supports two approaches of test case generation from CPN models. The **SSTCG** module on top of the state space tool of CPN Tools in Fig. 4.1 implements the state-space based test case generation approach. This approach is based on the computation of the state space and extraction of test cases by considering paths in the state space of the CPN model. The **SIMTCG** module on top of the simulation monitoring facilities of CPN Tools implements the simulation-based test case generation approach which is based on performing a simulation of the CPN model and extracting the test case corresponding to the execution path of the simulation.

The **CNF** (configuration) module is shared between the state space- and simulation-based test case generation. It provides configuration of the output directories and naming of test cases, and configuration of a *test case generation specification*. In order to

specify the observable input and output events during test case generation, the user needs to provide the test case specification by implementing a Standard ML structure conforming to the TCSPEC signature (interface) shown in Listing 4.1.

```
signature TCSPEC = sig
  val detection   : Bind.Elem -> bool;
  val observation : Bind.Elem -> TCEvent list;
  val format      : TCEvent   -> string
end;
```

Listing 4.1: Standard ML interface for test case specification.

This TCSPEC signature consists of three functions: a *detection function*, an *observation function*, and a *formatting function*. The Bind.Elem data type representing binding elements already exists in CPN Tools. The detection function constitutes a predicate that must evaluate to true for binding elements representing observable events. The observation function maps an observable binding element into an observable input or output event belonging to the TCEvent colour set. This function returns a list of observable events. The formatting function maps observable events into a string representation used to export the test cases into files. The detection and observation functions are specified independently of whether simulation-based or state space-based test case generation is employed. This makes it easy to switch between the two approaches. After providing these two functions, the tool first invokes the detection function on each arc of the state space (occurring binding element in a simulation) to determine if the corresponding event is observable, and if so, then the observation function is invoked to map the corresponding binding element into a representation of an observable event. We will give examples of these two functions for the two-phase commit protocol example in Section 4.2.1.

Given the test case specification provided by the user, the MBT/CPN tool can generate test cases and the user can control it by using the **Test Case Generation** module which implements the TCGEN signature (interface), partly shown in Listing 4.2. The ss function is used for state-space based test case generation. The sim function is used for simulation-based test case generation, and takes an integer as a parameter specifying the number of simulation runs to be executed for generating test cases. Both functions return a list of test cases. Each test case consists of a list of test case events TCEvent. The export function exports the test cases, based on the settings the user provided via the **CNF** configuration module. Finally, the **Export** module implements the export of the test cases into XML files.

```
signature TCGEN = sig
  val ss  : unit -> (TCEvent list) list;
  val sim : int -> (TCEvent list) list;
  val export : (TCEvent list) list -> unit
end;
```

Listing 4.2: Standard ML interface for test case generation.

4.2 Automated Model-based Testing

Based on the testing approach we have proposed in Fig. 3.2 in Section 3.3, we show below how automated model-based testing is performed with the aid of the MBT/CPN tool and an automatically generated test adapter. We explain our technique by using the CPN model of the two-phase commit (2PC) protocol, introduced in Section 1.5.

We first use the MBT/CPN tool to generate test cases from the 2PC CPN model. Then, we illustrate our technique to generate a test adapter, which can be used to execute test cases against an implementation of the coordinator process of the 2PC protocol implemented in the Go programming language. To do this, the workers module shown in Fig. 4.2 is used to obtain input events (stimuli) for the coordinator implementation, and the coordinator CPN module shown in Fig. 4.3 is used to obtain the expected outputs (test oracles) which in turn determine whether a test is successful or not. In that respect, the CPN module of the coordinator serves as an abstract specification of the coordinator process against which the behavior of the implementation can be compared.

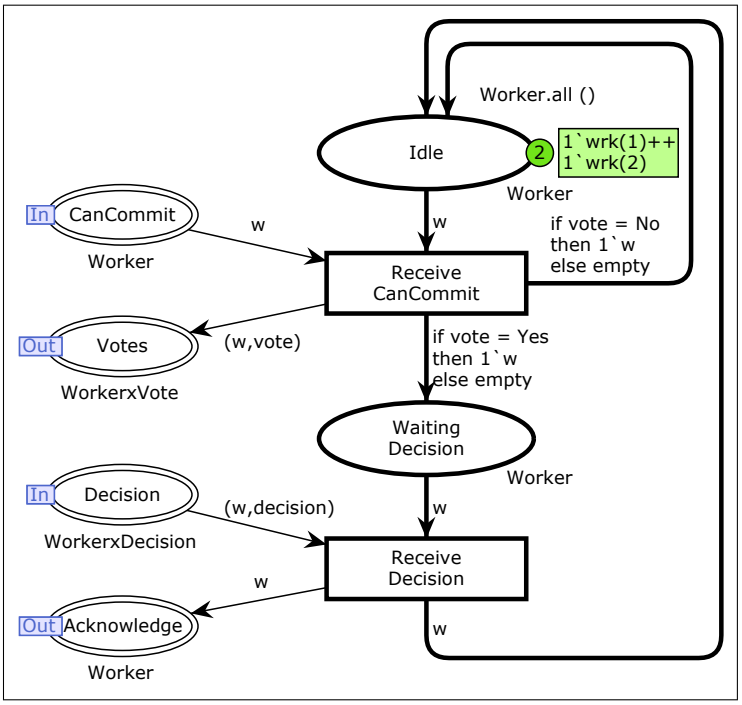


Fig. 4.2: Workers module of a 2PC CPN model.

4.2.1 *Test Case Generation*

As we have discussed in Section 4.1, the user of the MBT/CPN tool needs to extend the `TCEvent` base colour set by defining the colour sets `TCInEvent` and `TCoutEvent` according to the input and output events of the system to be observed. Therefore, for the 2PC protocol, the first step is to implement the input events to the coordinator implementation to be the votes of the individual workers. Output events are defined as the decisions sent to the individual workers by the coordinator and the overall decision to decide whether the transaction is to be committed or aborted.

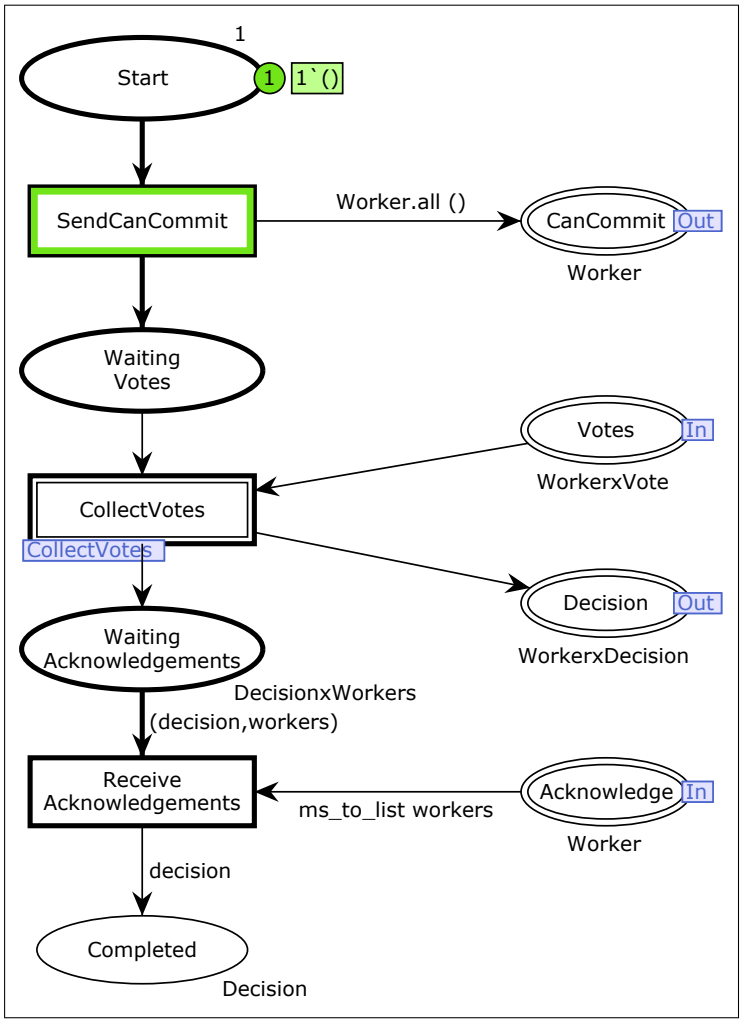


Fig. 4.3: Coordinator module of a 2PC CPN model.

```
val W = 2;
colset Worker  = index wrk with 1..W;     var w : Worker;
colset Workers = list Worker;             var workers : Workers;

colset Vote     = with Yes | No;          var vote : Vote;
colset Decision = with abort | commit;    var decision : Decision;

colset WorkerxVote      = product Worker * Vote;
colset WorkerxDecision  = product Worker * Decision;
```

Listing 4.3: Colour set and variable declarations for the 2PC protocol model.

Given the colour set definitions in the CPN model (Listing 4.3), the implementation of the `TCInEvent` and `TCOutEvent` colour sets is shown in Listing 4.4. The colour set `WDecision` indicates the decision sent to individual workers, while `SDecision` represents the overall system decision.

The second step is to define the test case generation specification `TCSPEC` by implementing the three functions specified in the signature (interface) in Listing 4.1. The

```
colset TCInEvent  = WorkerxVote;
colset TCOutEvent = union WDecision : WorkerxDecision +
                          SDecision : Decision;

colset TCEvent    = union InEvent  : TCInEvent +
                          OutEvent : TCOutEvent;
```

Listing 4.4: Definitions of the colour sets for observable events.

votes sent by individual serves as input events and can be obtained by considering occurrences of the ReceiveCanCommit transition in Fig. 4.2. The decisions of the coordinator in terms of output events can be obtained by considering the ReceiveDecision and ReceiveAcknowledgement transitions. The detection function is shown in Listing 4.5. For the 2PC protocol, the observation function accesses the values bound to the variables (`w`,`vote`, and `decision`) of the transitions and uses the constructors of the `TCEvent` and `TCOutEvent` data types to construct the observable events. The observation function can be implemented as shown in Listing 4.6. Finally, in order to export the test cases into an XML format, the user needs to provide a formatting function as part of the test case generation specification. The complete formatting function for the 2PC protocol is similar in complexity to the detection and the observation functions and has been omitted here.

```
fun detection (Bind.Workers'Receive_CanCommit _)  = true
 |  detection (Bind.Workers'Receive_Decision _)  = true
 |  detection (Bind.Coordinator'Receive_Acknowledgements _) = true
 |  detection _ = false;
```

Listing 4.5: The implementation of the detection function for the 2PC protocol.

```
exception obsExn;
fun observation (Bind.Workers'Receive_CanCommit (_,{w,vote})) =
     [InEvent (w,vote)]
 |  observation (Bind.Coordinator'Receive_Acknowledgements
     (_,{_,decision})) = [OutEvent (SDecision decision)]
 |  observation (Bind.Workers'Receive_Decision (_,{w,decision})) =
     [OutEvent (WDecision (w,decision))]
 |  observation _ = raise obsExn;
```

Listing 4.6: The implementation of the observation function for the 2PC protocol.

4.2.2 *Test Case Execution with a Generated Test Adapter*

To perform model-based testing using the test cases generated by MBT/CPN, the developer (tester) must either implement a test adapter or automatically generate it. Either way, the test adapter depends on the concrete SUT with which this test adapter needs to interact. However, a test adapter consists of the same overall components

independently of the SUT. We have developed a **Test Execution Engine** (shown in Fig. 4.4) which has an adapter generator to generate a test adapter and use it to perform test case execution against the SUT. To illustrate how MBT/CPN test cases can be used, and how the automation of our testing approach is performed for model-based testing, we outline how the **Reader** and **Tester** of a **Test Adapter** are generated for testing the Go implementation of the coordinator process. After these artifacts have been generated, we only need to provide the methods used to interact with the coordinator process in the generated **Tester** of the **Test Adapter**.

When exporting test cases from the MBT/CPN tool into an XML file (step (1) in Fig. 4.4), two elements of information are generated and inserted into the XML format. One element is test cases constructed according to the test case generation approach used and some basic system parameters of the SUT; the other element is the settings used to generate a **Test Adapter**. These settings include the names and types of variables for the input and expected output (oracle) of the SUT, and the names of method calls used by the **Tester** to interact with the SUT. The information included in these settings is used by the **Adapter Generator** of the **Test Execution Engine** which has predefined Go templates to generate a **Test Adapter** consisting of a **Reader** and a **Tester** in the Go programming language (step (2) and (3) in Fig. 4.4). After the **Test Adapter** has been generated, the **Reader** can read the information included in the test cases of the XML file (step (4) in Fig. 4.4). Then, the **Tester** can execute them against the **System Under Test** (SUT) and compare the output of the SUT with expected output (oracle) (steps (5), (6) and (7) in Fig. 4.4).

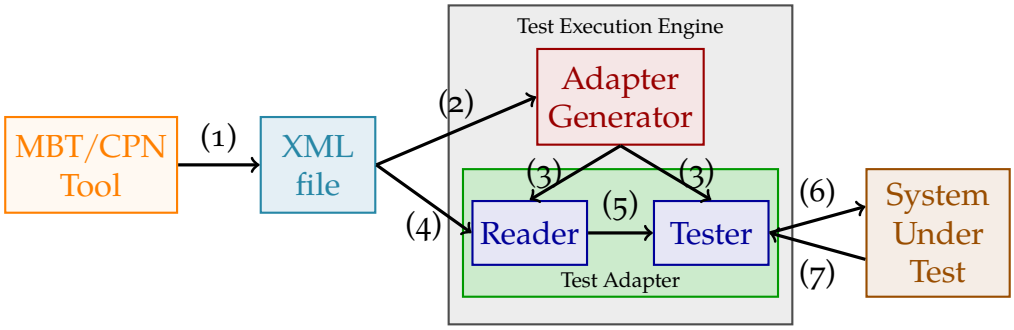


Fig. 4.4: Overall process of using the test execution engine.

The **Adapter Generator** is implemented in Go and its implementation relies on the *text/template* and *encoding/xml* packages, both belonging to the Go standard library. Given the *text/template* package, we have implemented predefined Go templates in the **Adapter Generator** which can be used to generate source code for the entire **Reader**, and components commonly used in the **Tester** independently of the SUT. The *encoding/xml* package makes it easy to define mappings between XML elements and Go structures. Go structures is the data structure in Go which can be used to store the setting information of a test adapter in the XML file generated together with test cases. The Go templates access this setting information stored in Go structures to generate a Test adapter.

Listing 4.7 shows an example of the setting information contained in the XML format used to generate a **Test Adapter** for testing the Go implementation of the coordinator process. The information within the <`TypeAssignments`> tag gives the

names and types of variables used in the predefined Go reader template to generate a **Reader** so that the generated **Reader** has these variables for reading test cases in the XML file. The names and types of variables are defined based on the definitions of variables in the Go implementation of the coordinator process, shown in Listing 4.8. The information within the <MethodCalls> tag specifies names of method calls used in the predefined Go tester template to generate the **Tester**. The generated **Tester** then has an interface (defined in the template) consisting of these method calls. The user only needs to implement methods specified in the interface to interact with the SUT.

```
<MethodCalls>
    <CallName>SendVote</CallName>
</MethodCalls>
<TypeAssignments>
    <SystemParameterTypes>
        <Field Name="NumberOfWorker" Type="int"></Field>
    </SystemParameterTypes>
    <InputType Name="VoteInput" Type="VoteInput">
        <Field Name="WorkerID" Type="WorkerID"></Field>
        <Field Name="VoteValue" Type="VoteEnum"></Field>
    </InputType>
    <OracleType Name="DecisionOracle" Type="DecisionSlice">
        <Field Name="WorkerID" Type="WorkerID"></Field>
        <Field Name="DecisionValue" Type="DecisionEnum"></Field>
    </OracleType>
    <OracleType Name="FinalDecision" Type="DecisionEnum"></OracleType>
</TypeAssignments>
```

Listing 4.7: Example XML format for the settings used to generate a test adapter.

```
type WorkerID int
type VoteEnum int
type DecisionEnum int
const (
   Yes VoteEnum = iota
   No
)
const (
   Commit DecisionEnum = iota
   Abort
)
type Vote struct {
   WorkerID  WorkerID
   VoteValue VoteEnum
}
type Decision struct {
   WorkerID      WorkerID
   DecisionValue DecisionEnum
}
type DecisionSlice []Decision
```

Listing 4.8: Variable declarations in Go for the coordinator process.

For testing the coordinator process, the generated **Tester** has the interface shown in Listing 4.9, and the definition of this interface in the predefined Go tester template is shown in Listing 4.10. The `MethodCallNames` variable is a string array that stores the names of method calls obtained from the <`MethodCalls`> tags in XML format (as shown in Listing 4.7). The range action initializes a variable (`callname`) which is set to the successive elements of the iteration of the array `MethodCallNames`. Therefore, any name specified by the <`MethodCalls`> tags in XML format can be generated as a method call

```
type systemTest interface {
    Start()                  // method to start system
    SendVote(v *TestValue)   // method to interact with system
    OutputChecker(cs *SystemTestCases) // method to check output
}
```

Listing 4.9: Interface in the Tester module generated in Go programming language.

```
type systemTest interface {
    Start()
{{- range $callname := .MethodCallNames }}
{{$callname}}(v *TestValue)
{{- end }}
    OutputChecker(cs *SystemTestCases)
}
```

Listing 4.10: The definition of the interface in the Go tester template.

in the interface in the **Tester**. For testing the coordinator process, only one method call `SendVote` is obtained from the settings specified in XML format in Listing 4.7 in order to interact with the coordinator. The `Start()` and `OutputChecker(cs *SystemTestCases)` methods in Listing 4.9 are default methods specified in the predefined Go tester template in Listing 4.10. The user of a generated **Test Adapter** must implement these method calls specified in the interface of the **Tester**.

After both the **Reader** and the **Tester** have been generated, the **Reader** can read test cases specified in the XML file, as shown in Listing 4.11. In addition to the input values and oracles specified in test cases, the **Reader** also reads the names of method calls which will be executed by the generated **Tester** to interact with the SUT. The **Tester** will know if these method calls are executed concurrently or sequentially, as indicated by the <`Concurrent`> tag. That is, the names of method calls specified within the <`Concurrent`> tag indicate that these methods need to be executed concurrently by the **Tester**. Otherwise, these method calls will be executed sequentially.

An example is considered in Listing 4.11. The method call `SendVote` is specified three times by the <`Call Name="SendVote"`> tag within the <`Concurrent`> tag. This means that the generated **Tester** will execute the `SendVote` method three times concurrently, initialized by three workers (defined by <`NumberOfWorker`> tag), respectively. Based on the implemented Go template, the generated **Tester** has an executor called `funcExecutor` (shown in Listing 4.12) to execute the method calls specified by the `Call Name` tag in the XML format. Line 8 and Line 12 in Listing 4.12 indicate that the **Tester** can call any method with the name stored in the variable `funcName` (such as `SendVote`). This `funcExecutor` function can be invoked either concurrently or sequentially in the generated **Tester**. To interact with the coordinator process, the `funcExecutor` function is invoked by three *go routines* (threads in Go) concurrently to send three `Vote` messages, respectively, to the coordinator process. After the coordinator has handled the `Vote` messages, the `OutputChecker(cs *SystemTestCases)` in Listing 4.9 compares the testing results it received from the coordinator with the oracles (`Decisions` and `FinalDecision`). To do this, the generated **Tester** uses the *testing* package from the

Go standard library. Go's testing infrastructure comprises the `go test` command which allows us to simply run and execute our generated tests and obtain pass/fail information for each test case execution. The Go testing infrastructure also includes a tool which can be used to evaluate our approach by measuring the statement coverage.

```
<SystemParameters>
    <NumberOfWorker>3</NumberOfWorker>
</SystemParameters>
<TestCase ID="1">
    <InputValues>
        <Concurrent>
            <Call Name="SendVote">
                <InputValue>
                    <VoteInput>
                        <WorkerID>1</WorkerID>
                        <VoteValue>0</VoteValue>
                    </VoteInput>
                </InputValue>
            </Call>
            <Call Name="SendVote">
                <InputValue>
                    <VoteInput>
                        <WorkerID>2</WorkerID>
                        <VoteValue>0</VoteValue>
                    </VoteInput>
                </InputValue>
            </Call>
            <Call Name="SendVote">
                <InputValue>
                    <VoteInput>
                        <WorkerID>3</WorkerID>
                        <VoteValue>0</VoteValue>
                    </VoteInput>
                </InputValue>
            </Call>
        </Concurrent>
    </InputValues>
    <Oracles>
        <DecisionOracle>
            <WorkerID>1</WorkerID>
            <DecisionValue>0</DecisionValue>
        </DecisionOracle>
        <DecisionOracle>
            <WorkerID>2</WorkerID>
            <DecisionValue>0</DecisionValue>
        </DecisionOracle>
        <DecisionOracle>
            <WorkerID>3</WorkerID>
            <DecisionValue>0</DecisionValue>
        </DecisionOracle>
        <FinalDecision>0</FinalDecision>
    </Oracles>
</TestCase>
```

Listing 4.11: Example XML format for a test case.

```
1  // funcExecutor executes different method calls given in XML tags.
2  func funcExecutor(t *testing.T, tester interface{}, funcName string, params ...
       interface{}) {
3      t.Helper()
4      inputArgs := make([]reflect.Value, len(params))
5      for i, param := range params {
6          inputArgs[i] = reflect.ValueOf(param)
7      }
8      fn := reflect.ValueOf(tester).MethodByName(funcName)
9      if !fn.IsValid() {
10         t.Errorf("method '%s' not found", funcName)
11     }
12     fn.Call(inputArgs)
13 }
```

Listing 4.12: The funcExecutor in Tester generated in Go programming language.

4.3 Results and Contributions

The main contribution of article [134] is to present our approach to model-based testing using CPNs and the supporting MBT/CPN tool. MBT/CPN is implemented on top of CPN Tools to support test case generation from CPN models. MBT/CPN has been developed as part of our research into MBT for quorum-based distributed systems and protocols [132, 133]. The main idea underlying our approach is for the modeler to capture the observable input and output events (transitions) in a test case specification. A main feature of the MBT/CPN tool is the uniform support for state space and simulation-based test case generation.

A second contribution of our work on MBT/CPN is the experimental evaluation of the tool on a two-phase commit transaction protocol implemented using the Go programming language, with the coordinator as the SUT. The lines of code for the coordinator is around 120 lines. We have used statement coverage (supported by the Go tool chain) as the quantitative evaluation criteria of the test cases generated by our approach.

Table 4.1 illustrates experimental results for the two-phase commit protocol with different number of workers **W**, after applying our approach. The **Gen** column specifies the approach used for test case generation, either state space (SS) or simulation (SIM). The **Size/Steps** column gives the size of the state space (with the number of nodes / arcs) or the number of steps in a simulation run. The **Test Cases** column records the number of test case generated, and the **Time** column gives the total time (in second) used for test case generation. Finally, the **Coverage** column gives the statement coverage obtained for the coordinator implementation.

For the test case generation with the simulation based approach, we stopped increasing the number of steps in the simulation after it reached the same number of test cases as the state-space-based generation approach that displays the maximum number of test cases which can be obtained. Specifically, in Table 4.1, as **W** increases, more simulations are needed to reach the maximum number of test cases. Also, for state space based test case generation, we did not pursue the experiments beyond four

Table 4.1: Experimental results for the two-phase commit protocol.

W	Gen	Size / Steps	Test Cases	Time	Coverage
2	SS	59 / 86	4	<1	94.7 %
2	SIM	5	3	<1	84.2 %
2	SIM	10	4	<1	94.7 %
3	SS	357 / 614	8	<1	94.7 %
3	SIM	10	4	<1	94.7 %
3	SIM	20	8	<1	94.7 %
4	SS	2,811 / 5,957	16	5	94.7 %
4	SIM	50	13	<1	94.7 %
4	SIM	100	16	<1	94.7 %
5	SIM	100	31	<1	94.7 %
5	SIM	200	32	<1	94.7 %
10	SIM	5000	1,015	13	94.7 %
10	SIM	10000	1,024	25	94.7 %
15	SIM	10000	8,627	91	84.2 %
15	SIM	20000	14,946	265	94.7 %

workers since it became quite time-consuming to generate test cases from larger state spaces. However, the simulation based approach can easily handle configurations with 5, 10, and 15 workers, as shown in Table 4.1. This demonstrates the scalability of simulation based test case generation.

The results show that we obtain 94.7 % statement coverage with the test cases generated by both the state space and the simulation based approaches. Due to the implementation of the coordinator containing error handling code, which is not covered by the generated test cases (as any failures are not part of the model), the results of statement coverage cannot reach 100 % The results also show that the statement coverage for both **SIM-**5 and **SIM-**10000 is 84.2 %, since the simulation based approach did not succeed to cover all the possible executions of the CPN model in the absence of guided search.

We have also applied MBT/CPN to a distributed storage protocol [132] and the Paxos distributed consensus protocol [133] as discussed in Chapter 3. The distributed storage protocol and the Paxos protocol were both implemented in the Go programming language using a quorum-based distributed systems middleware [87]. These experiments also showed a high statement coverage and demonstrated in addition that our approach is able to detect programming errors via the generation and execution of unit and system tests.

4.4 Related Work

There exist a wide variety of model-based testing tools for test case generation. The Conformiq Qtroniq [63] tool derives functional test cases from a system model, and generates test cases with expected outputs either online or offline using a symbolic execution algorithm. The generated test cases are mapped into the TTCN-3 format. The Automatic Efficient Test Generation (AETG) [34] tool is aimed at efficient generation

of test cases by decreasing the amount of test data required for the input test space. However, the test oracles have to be furnished manually. Tretmans et al. [122] have presented the TorX tool. This tool can generate test cases based on a random walk through to the state space. The test cases can be generated either offline or on-the-fly during the test execution. In TorX, an adapter component is used to translate the inputs to a form readable by the SUT and to check the actual outputs from the SUT against expected outputs.

A model-based testing tool known as the Integration and System Test Automation (ITSA) tool [137] follows a CPN-based approach to generate test cases for a variety of languages including Java, C/C++, C# and HTML. Compared to the ITSA tool, our model-based testing approach is not tied to a particular programming language, since the test cases are generated in an XML format, which can be read by any programming language. The ITSA tool uses the state space of the testing model to generate and select test cases. For our model-based testing approach with CPN models, we can perform both simulation-based and state-space-based test case generation. To obtain concrete test cases with input data, the ITSA tool relies on a separate model-to-implementation mapping. In contrast, we obtain the input data for the SUT and the method calls directly from the data modeling contained in the CPN models. As a case study, the ISTA tool has been applied to an online shopping system. However, their approach does not appear to be suitable for testing complex distributed systems and protocols, since they do not consider and address concurrency and failures, which is at the core of our work.

Test code generation with timed event-driven CPNs has been used by Faria et al. [44]. Instead of using CPNs as a direct interface to the user, the authors generate test cases from UML sequence diagrams. Their tool suite has a different focus in that they instrument a running system to observe the messages specified in the sequence diagrams. The toolset has not been used for unit and system testing of distributed systems, but can only perform JUnit tests on Java-based applications. Liu et al. [89] has also proposed a CPN-based test generation approach. This approach requires a conformance testing-oriented CPN (CT-CPN) model and a PN-ioco relation to be defined. A PN-ioco relation specifies how an implementation conforms to its specifications. This approach uses a simulation-based test case generation algorithm for the CT-CPN model. In our approach, test cases can be generated directly using simulation-based or state space-based test case generation for an existing implementation of the SUT.

A model-based test generation technique based on CPNs is used by Wu, Schnieder, and Krause [136] to verify a module of a satellite-based train control system. The authors use CPN Tools to generate the reachability graph of the test model and then use state space analysis with CPN Tools to extract the expected output of each test case from the path of the graph. However, their approach does not support simulation-based test case generation, which is important for scalability. Farooq, Lam and Li has proposed a test sequence generation technique [45]. For this technique, a CPN model is derived from a UML Activity Diagram, and the derived model is then used to generate test sequences. The authors demonstrate their approach on a fictional enterprise commerce system, describing the process of purchasing products online. In our model-based testing approach, we design a testing framework consisting of the constructed CPN test models, test case generation algorithms and test adapters, in order to enable the

execution of the generated tests. We have furthermore performed evaluation of our approach on real distributed systems and protocol implementation in the form of a distributed storage service [132] and a Paxos distributed consensus protocol [133].

CHAPTER 5

PATH COVERAGE VISUALIZATION AND MULTI-OBJECTIVE SEARCH WITH MODBAT

Software testing is a widely used, scalable and efficient technique to discover software defects [100]. However, generating sufficiently many and diverse test cases to obtain good coverage of the software system under test is challenging. Especially for complex software systems, it is infeasible to explore and generate all the possible test data for the software system under test. It is therefore important to use test adequacy criteria to measure and evaluate the extent to which sufficient test cases have been generated and executed against the SUT. Also, effective test case generation approaches are required to generate sufficiently many and diverse test cases to obtain desired results relative to the test adequacy criteria.

This chapter summarizes the articles [128, 129] included in Part II. Article [128] focuses on execution path coverage which can be considered as a test adequacy criterion that assesses the degree to which the testing model has been explored. We present a technique to measure and visualize execution path coverage of test cases in the context of model-based software systems testing. Two types of visualizations for path coverage, so-called *state-based graphs (SGs)* and *path-based graphs (PGs)*, are proposed based on visualization abstractions. The visualization feedback provided by these two types of simplified graphs is useful to understand to what degree the model and the SUT are executed by the generated test cases, to understand execution traces, and to locate weaknesses in the coverage of the model. To evaluate our approach, we have performed experiments on a collection of examples, including the ZooKeeper distributed coordination service.

Article [129] presents a search-based approach relying on multi-objective reinforcement learning and optimization for test case generation. We propose and implement a bandit heuristic search strategy [129] in the Modbat tester to tackle the exploration versus exploitation dilemma faced by test case generation. We use four test adequacy criteria as objectives and apply multi-objective optimization to tune our search strategy to get optimal test suites while considering the trade-offs of the chosen test adequacy criteria. The multi-objective optimization has been performed with the aid of the jMetal multi-objective optimization framework [41, 101] and the NSGA-II genetic algorithm [39]. We have experimentally evaluated our approach on a collection of examples including the ZooKeeper distributed coordination service and the PostgreSQL database system, to compare our bandit heuristic search strategy with the random approach for test case generation.

We have implemented our execution path coverage visualization and bandit heuristic search strategy as new features of the Modbat 3.4 release[2]. Both of them consider that test adequacy criteria are important for MBT to give developers and testers confidence about how well the models and the SUT are explored by the generated test cases.

In this chapter, we first discuss our approach to capture execution paths and the abstractions proposed for path coverage visualization of Modbat models. We then present our bandit heuristic search strategy and multi-objective optimization using jMetal and the NSGA-II algorithm. At the end of this chapter, we summarize our conclusions and discuss related work.

5.1 Extended Finite State Machines (EFSMs)

Path coverage is a test adequacy criterion that helps testers to determine if the software has been adequately exercised by a test suite. Path coverage concerns a sequence of branch decisions instead of only one branch at a time as in branch coverage. It is known to be hard to reach 100% path coverage since the number of execution paths usually increases exponentially with each additional branch or cycle [86].

For visualizing path coverage, we need to capture the execution paths. In this section, we first give the definition of *extended finite state machines (EFSMs)*. EFSMs is the underlying theoretical foundation used in Modbat to express models, and hence it is also the foundation for our definition of *execution paths* and test cases.

Definition 1 (Extended Finite State Machine)**.** *An* extended finite state machine *is a tuple* $M = (S, s_0, V, A, T)$ *such that:*

- S *is a finite set of states, including an initial state* s_0*.*
- $V = V_1 \times \ldots \times V_n$ *is an* n*-dimensional vector space representing the set of values for variables.*
- A *is a finite set of actions* $A : V \to (V, R)$*, where* $\mathtt{res} \in R$ *denotes the* result *of an action, which is either* successful, failed, backtracked, *or* exceptional*. A successful action allows a test case to continue; a failed action constitutes a test failure and terminates the current test; a backtracked action corresponds to the case where the enabling function of a transition is* false *[29]; exceptional results are defined as such by user-defined predicates that are evaluated at run-time, and cover the non-deterministic behavior of the SUT. We denote by* $\mathsf{Exc} \subset R$ *the set of all possible exceptional outcomes.*
- T *is a transition relation* $T : S \times A \times S \times E$*; for a transition* $t \in T$ *we denote the left-side (origin) state by* $s_{\mathtt{origin}}(t)$ *and the right-side (destination) state by* $s_{\mathtt{dest}}(t)$*, and use the shorthand* $s_{\mathtt{origin}} \to s_{\mathtt{dest}}$ *if the action is uniquely defined. A transition includes a possible empty mapping* $E : \mathsf{Exc} \to S$*, which maps exceptional results to a new destination state.*

Definition 1 is different from the definition of a standard EFSM [30] since we merge the enabling and update functions into a single action $\alpha \in A$, and handle inputs and outputs inside the action. Actions concern preconditions, inputs, executing test actions on the SUT, and its outputs. An action may also include assertions; a failed

assertion causes the current test case to fail. In addition to that, transitions also support non-deterministic outcomes in our definition.

Based on Definition 1, a finite *execution path* can be defined as a sequence of transitions starting from the initial state and leading to a *terminal state*. A terminal state is a state after a test failed, a state where no transitions are enabled, or a state where exploration of the state space has been truncated. Each finite execution path represents a test case that can be generated from a model. That is, a test case is an execution path consisting of a sequence of transitions. The execution path of the model is formally defined as

Definition 2 (Execution Path). *Let* $M = (S, s_0, V, A, T)$ *be an EFSM. A* finite execution path p *of* M *is a sequence of transitions, which constitute a path* $p = t_0 t_1 \dots t_n, \quad t_n \in T$, *such that* $s_{origin}(t_0) = s_0$, *the origin and destination states are linked:* $\forall i, 0 < i \leqslant n, s_{origin}(t_i) = s_{dest}(t_{i-1})$, *and* $s_{dest}(t_n) \in S_{terminal}$; $S_{terminal}$ *is the set of terminal states.*

5.2 Representation of Execution Paths

To visualize path coverage, we need to record each executed path of a test suite. When Modbat generates test cases based on its models, we record the paths executed by the test cases in a *trie*. A trie is a prefix tree data structure where all the descendants of a node in the trie have a common prefix. We store information related to an executed transition into each trie node n. This information includes the following: executed transition t; transition information ti; transition repetition counter trc to count the number of times that transition t has been executed repeatedly without any other transitions executing in between during a test-case execution (no repetition when $trc = 1$); transition path counter tpc to count the number of paths that have this transition t executed trc times in a test suite; the set of children Ch of node n; and a Boolean variable lf to decide if the current node is a leaf of the tree.

The transition information ti consists of the $s_{origin}(t)$ and $s_{dest}(t)$ states of the transition, a transition identifier tid, a counter cnt to count the number of times this transition is executed in a path, an action result res, which could be successful, backtracked or failed, and a sequence of transition-internal choices C for modeling non-determinism.

Fig. 5.1 shows an example of a trie data structure representing the following three execution paths executed by a test suite:

- $p_0 = [a \to b, b \to b, b \to c, c \to d]$,
- $p_1 = [a \to b, b \to b, b \to b, b \to c, c \to d]$,
- $p_2 = [a \to b, b \to b, b \to e]$,

where a, b, c, d, and e are states. The node labeled root represents the root of the trie in Fig. 5.1. It should be noted that this data structure is not a direct representation of the execution paths and it is not the trie data structure that we eventually visualize in our approach. Each non-root node in the trie has been labeled with the transition it represents. For instance, node 1 represents the transition $a \to b$ and node 2 represents the transition $b \to b$. It can be seen that all the three execution paths stored in the

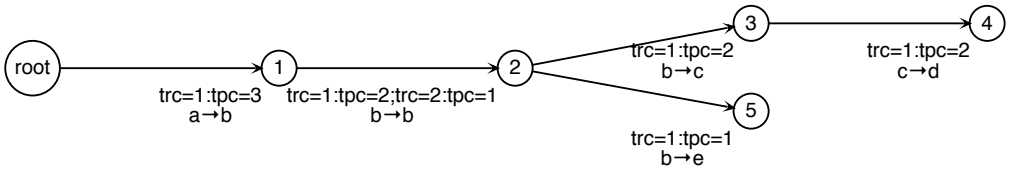


Fig. 5.1: Example trie data structure representing three executed paths.

trie have $a \rightarrow b$ followed by $b \rightarrow b$ as a (common) prefix. The label attached to each non-root node represents the transition counters associated with the node (the value of the trc is shown before the colon; the value of the tpc is shown after the colon). For example, the transition $b \rightarrow b$ associated with node 2 has been taken three times in total. Two paths, (p_0 and p_2) execute this transition once (label trc=1:tpc=2), while one path p_1 executes it twice (label trc=2:tpc=1). In each node of the trie, we use a mapping $\langle tid, res \rangle \mapsto n$ to connect a parent node and a child node.

5.3 Path Coverage Visualization

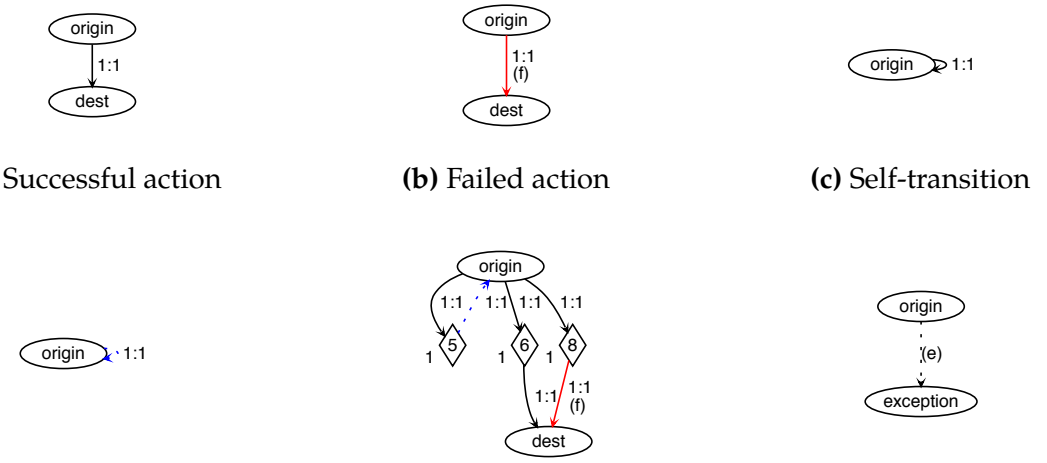
We have implemented a path coverage visualizer as a tool for Modbat so that Modbat can use our approach to first capture executed paths and then visualize them. The path coverage visualizer is presented in the article [128] and can produce two types of directed and abstracted graphs: state-based graphs (SGs) and path-based graphs (PGs) based on information stored in the trie data structure representing the executed paths. The SGs convey the structure and behavior of the model well, since they give an overview of the models based on the EFSM with detailed information about how and which states and transitions have been executed; while the PGs show distinct executed paths well, without providing detailed information of states, but they can directly show how many linearly independent paths are executed.

Below, we first introduce the basic visualization elements underlying the construction of the SGs and PGs. Then, we discuss the abstractions we use to reduce the complexity of the graphs in order to visualize execution paths.

5.3.1 *Basic Visualization Elements*

The basic visualization elements used to construct the SGs and PGs include node and edge styles of graphs (shape, color and thickness) to indicate different features of the path coverage visualization. Fig. 5.2 and Fig. 5.3 illustrate these basic visualization elements of the SGs and PGs, respectively.

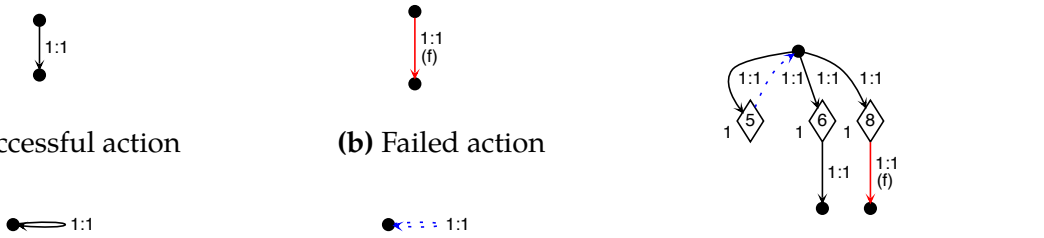
For node styles, we use three types of node shapes in the graphs for path coverage visualization; *elliptical nodes*, *point nodes*, and *diamond nodes*. We use elliptical nodes to represent states in the SG as shown in Fig. 5.2, while we use point nodes as the connections between transitions (or steps of transitions as introduced Section 1.4) in the PG shown in Fig. 5.3. Diamond nodes are used to visualize internal choices in both



(a) Successful action **(b)** Failed action **(c)** Self-transition

(d) Backtracked transition **(e)** Action with choices **(f)** Exception result

Fig. 5.2: Basic visualization elements of the state-based graphs (SGs).



(a) Successful action **(b)** Failed action

(c) Self-transition **(d)** Backtracked transition **(e)** Action with choices

Fig. 5.3: Basic visualization elements of the path-based graphs (PGs).

the SGs and PGs, as shown in Fig. 5.2e and Fig. 5.3e. A value is attached inside each diamond node to show the value associated with the choice. An optional counter value label with the format $n : m$ next to each diamond node shows the number of times this choice was taken.

In addition to node styles, we also apply different directed edge styles in both the SGs and PGs to represent different outcomes of executed transition actions (introduced in Section 1.4) as stored in the trie structure. We use different kinds of edge shapes and color styles to distinguish the action results of transitions. Specifically, black solid edges are used to represent successful transitions (Fig. 5.2a) and Fig. 5.3a). Blue dotted edges are used to indicate transitions that are backtracked (Fig. 5.2d and Fig. 5.3d). Red solid edges with a (f) label are used to visualize failed transitions (shown in Fig. 5.2b and Fig. 5.3b). Black solid loops represent self-transitions ($s_{origin}(t) = s_{dest}(t)$) Fig. 5.2c and Fig. 5.3c). Black dotted edges labeled (e) are used to represent exceptional results for the SGs (shown in Fig. 5.2f), while for the PGs, such type of edge is ignored by merging the point nodes of $s_{origin}(t)$ and the exceptional destination state of a transition t into one point node. When a transition t has multiple *steps* (Fig. 5.2e and Fig. 5.3e), we only apply the edge styles to the last step connecting to $s_{dest}(t)$, while other step edges use a black solid style.

Each edge use thickness as an attribute to indicate how frequently a transition is executed for the entire test suite. That is, the thicker an edge is, the more frequently is the transition executed. We define the thickness of an edge as $\ln(\frac{\sum count*100}{nTests} + 1)$, where `nTests` is the total number of executed test cases; the value of `count` is the `tpc` value of a transition in each path if there are no internal choices for this transition. However, if a transition has internal choices, then we use the value of the counter for each internal choice as the value of `count`. Since we merge edges in the graphs representing the same transition or corresponding to the same choice from different paths, we then compute $\sum$ `count` in two ways: 1) accumulate all values of `counts` obtained for the same transition that dose not have any choice; or 2) accumulate all values of counters obtained for the same internal choices if a transition has internal choices.

A label can optionally be attached to an edge to give additional information, such as transition identifier `tid`, and values of the counters `trc` and `tpc` (with the format `trc : tpc`). In both Fig. 5.2 and Fig. 5.3, the values of counters are all 1 : 1 which indicates that each transition in a test case is executed only once without any repetitions, and there is only one path that has this transition executed.

5.3.2 *State-based and Path-based Graphs*

Our state-based (SG) and path-based (PG) graphs rely on abstractions that we proposed in the article [128] to form the foundation of our approach to visualize path coverage. These abstractions reduce the complexity of the representation of the execution paths. We use the definition of *linearly independent path* [70] as a foundation to reduce the complexity of graphs. A linearly independent path is any path through a program that contains at least one new edge which is not included in any other linearly independent paths. Any path having a new node compared to all other linearly independent paths, is also linearly independent. The reason for this is because having a new node automatically implies having a new edge. This means that a path that is a subpath of another path is not considered to be a linearly independent path. A subpath q of an execution path p is a subsequence of p (possibly p itself), and an execution path p traverses a subpath q if q is a subsequence of p. When visualizing execution paths, we merge subpaths from different linearly independent paths in both the SGs and PGs with the aid of the trie data structure.

5.3.2.1 *State-based Graphs*

An SG is an abstracted graph which 1) reduces the amount of redundant edges representing the same transition/step between two states; 2) reduces the redundant choice nodes having the same choice value. In general, these redundant edges and nodes contribute to making the graph large, complex, and difficult to analyze, especially when the systems to be analyzed are also large and complex.

In order to address the complexity of graphs, we propose abstractions which reduce redundancy and obtain abstracted graphs. Here, we use the ChooseTest Modbat model with 1000 executed test cases as an example to show how the SG is obtained using our four abstraction steps:

1. *Merge edges of subpaths*: when storing transitions in the trie, we also use the trie data structure to merge subpaths of linearly independent paths. As discussed in Fig. 5.1, transition $a \to b$ and $b \to b$ are (common) prefixes for all the three execution paths p_0, p_1 and p_2. All three execution paths traverse the subpaths $a \to b$ and $b \to b$. Therefore, to obtain the SG, we merge edges representing transition $a \to b$ and $b \to b$ from the three execution paths into one edge by the trie data structure. Then, an edge label of the form "$trc : tpc$" may be used to show how a transition represented by this edge is executed. After merging edges of subpaths, we only have linearly independent paths in the graph as shown in Fig. 5.4. There are seven linearly independent paths in this case: $p_0 = [ok \to end]$, $p_1 = [ok \to ok, ok \to end]$, $p_2 = [ok \to ok, ok \to err(backtracked), ok \to end]$, $p_3 = [ok \to ok, ok \to err]$, $p_4 = [ok \to err(backtracked), ok \to end]$, $p_5 = [ok \to err(failed)]$ and $p_6 = [ok \to err]$.

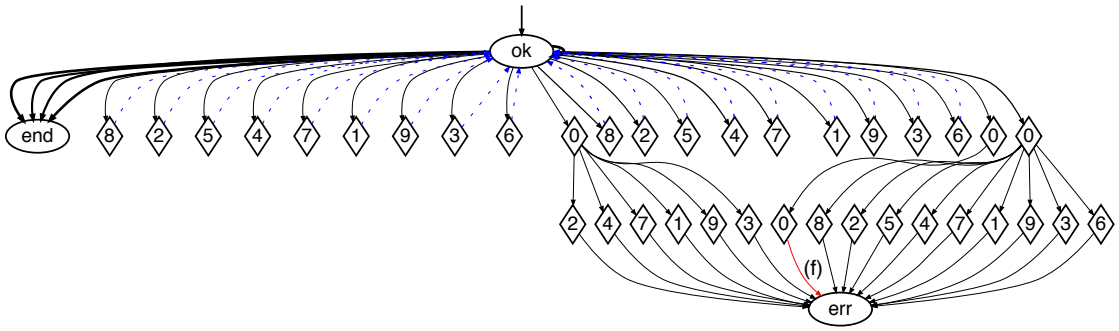


Fig. 5.4: The graph for 1000 test cases of ChooseTest after merging subpaths.

2. *Merge edges of linearly independent paths*: it can be noticed that a graph such as Fig. 5.4 may still have redundant edges between two states that represent the same transition with the same action result from different linearly independent paths, also after merging edges of subpaths.

   For instance, the four edges from the "ok" to the "end" states, originates from the four linearly independent paths: p_0, p_1, p_2 and p_4. Therefore, we merge such edges into one single edge and aggregate the path coverage counts. It is optional to use an edge label to show the aggregated counts on the form "$trc : tpc$", using ";" as the separator, e. g., "$1 : 304; 1 : 158; 1 : 177; 1 : 290$" for the edge between the "ok" and "end" states after merging p_0, p_1, p_2 and p_4.

3. *Merge internal choice nodes*: we merge redundant choice nodes of a transition in two ways. First, when we store transitions in the trie in Step 1, we merge choice nodes from different choice lists recorded as information for each transition, if these choice nodes have the same choice value, and they are a (common) prefix of choice lists. For example, for choice lists $[0, 1, 2]$ and $[0, 1, 3]$ (0, 1, 2, 3 are choice values), both of them have choice nodes with values 0 and 1 which together become a (common) prefix $[0, 1]$ for these two lists. Then, the choice nodes with

values 0 and 1 are merged to become one choice node, respectively, when we store transitions in the trie.

Second, if there are still redundant choice nodes of a transition with the same value appearing more than once and from different linearly independent paths, such as the choices in Fig. 5.4, then we merge them into one choice node during Step 2. For both approaches, we get the total number of times a choice value appears in the SG, and this number can be shown by an optional label of the final choice node after merging.

4. *Merge loop edges*: we merge loop edges representing self-transition loops and backtracked transitions; they are merged if they represent the same transition with the same action result.

Fig. 5.5 illustrates the final SG with all abstractions for the ChooseTest model.

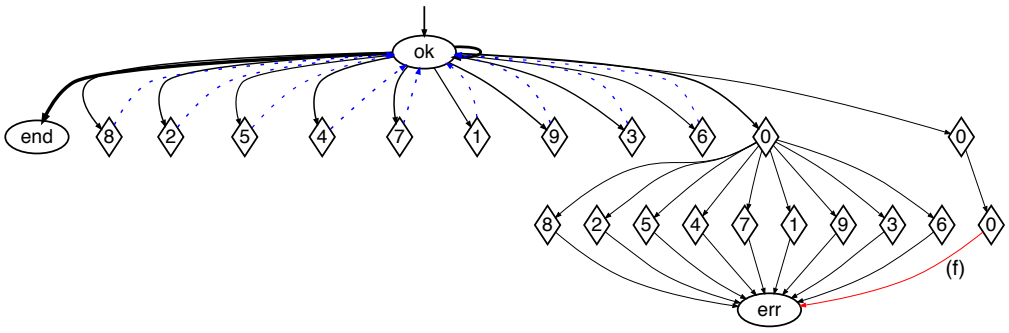


Fig. 5.5: SG for 1000 test cases of ChooseTest with all abstractions applied.

5.3.2.2 *Path-based Graphs*

The PG is a directed graph constructed with nodes and directed edges. The nodes include point nodes and diamond nodes. Each node has an identifier. Point nodes represent connections between transitions, and each diamond node represents a choice and its associated value. The edges represent transitions and steps, and are connected by point nodes or diamond nodes according to their identifiers, which results in constructing paths one by one. All constructed paths start with the same initial point node and end in different final point nodes. The number of constructed paths in the PG indicates the number of linearly independent paths. Note that, in contract to SG, each node in a PG does not correspond to a state in the EFSM, but corresponds to a step in a linear independent path through the EFSM.

A PG is an abstracted graph obtained by applying the abstractions proposed in the article [128]. As was the case for SG, abstractions are used to address the complexity of the PGs. Three abstractions proposed in the article [128] are applied for PGs:

1. Merge edges of subpaths using the same approach as for Step 1 of the SG.

2. Merge internal choice nodes with the first approach of Step 3 used to merge choice nodes for the SG.

3. Merge loop edges representing self-transition loops and backtracked transitions as for the SG.

We do not merge edges of linearly independent paths for PGs as we do for the SGs, since the goal for the PGs is to show linearly independent path coverage after using the abstractions.

Fig. 5.6 shows the resulting PG for the ChooseTest model after application of abstractions. Seven black final point nodes can be clearly seen from seven paths, and they indicate that seven linearly independent paths have been executed. The information about the number of linearly independent paths is one characteristic of the PG, and this information cannot easily be derived from the SG shown in Fig. 5.5. We provide further examples of visualization in Section 5.5.

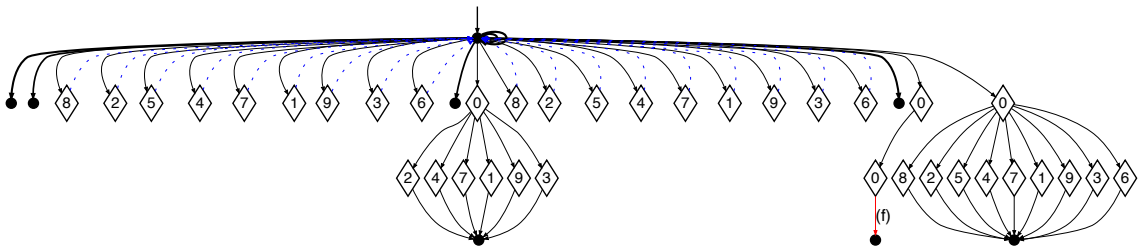


Fig. 5.6: Path-based graph for 1000 test cases of the ChooseTest model.

5.4 Multi-objective Search

MBT relies on the automatic generation and execution of test cases. Before generating test cases and executing them, it is necessary to choose *test adequacy criteria* aimed at assessing the quality of the generated test cases. These criteria may consider the discovery of defects and code coverage in general. To evaluate if the generated test cases are sufficient, we use test adequacy criteria including state and transition coverage, linearly independent path coverage, and the number of test cases needed to find the first failure.

Usually, it is a challenge for MBT to obtain good results on test adequacy criteria with a small test suite having few redundant test cases generated. Also, random test case generation approaches are often uncontrolled and might result in test suites having redundant test cases which cover only few execution paths of the model and the SUT.

For test case generation with Modbat, test cases are derived from Modbat models, and each test case represents one execution path which in turn consists of a sequence of transitions. The generation of a test case therefore relies on the decisions made in each step to select the transitions that are to be part of the constructed execution paths. The decision made to select a transition for test case generation faces the exploration

versus exploitation dilemma in terms of finding a balance between: a) the exploration of different transitions which have not been selected; or have been selected fewer times, but might result in better addressing the test adequacy criteria; and b) the continuous exploitation of the selected transitions which have empirically resulted in better outcomes (e. g., a high coverage).

In order to address this dilemma, we focus on the test case generation with a search-based approach based on multi-objective reinforcement learning and optimization. The goal is to find and generate a subset of test cases that optimizes the results on chosen test adequacy criteria while considering their trade-offs. Specifically, we consider that 1) the process of test case generation faces the exploration versus exploitation dilemma when exploring possible test cases; 2) obtaining good and balanced results for the chosen test adequacy criteria with fewer generated test cases is a multi-objective optimization problem.

5.4.1 *Bandit Heuristic Search for Test Case Generation*

Reinforcement learning [121] is the subfield of machine learning devoted to studying problems and designing algorithms that analyze the exploration versus exploitation dilemma. A well-established class of sequential decision problems in this context is the multi-armed bandit problem which has been extensively studied by Berry and Fristedt [18].

The basic idea of Bandit problems involves the multi-armed bandit slot gambling machine in a casino, and a gambler (agent) needs to choose among K arms (actions) in I rounds on this gambling machine. This gambler wants to maximize the cumulative money (reward) as much as possible by consistently choosing the optimal arm over rounds [23]. During each round $i = 1, \ldots, I$, the gambler selects an arm $j \in \{1, \ldots, K\}$ and receives the reward money $r_{(j,i)}$. The gambler has a goal: on one hand, finding out (exploit) which arm could be currently optimal and have the highest expected reward money; on the other hand, exploring the reward money of other arms that currently are not optimal, but may turn out to be optimal in the long run [15, 23, 76].

In the article [129], we have used the UCB1 bandit algorithm from the UCB family of algorithms [15] as a basis for implementing our multi-objective search strategy for test case generation. Based on the UCB1 algorithm proposed in [15], for generating test cases, we consider each transition $t_j \in T$ to select for constructing an execution path (a test case) as a bandit arm to play. We denote the reward function as $r : T \to \mathbb{R}$. After executing a transition $t_j \in T$, its corresponding immediate reward $r_{t_j} \in \mathbb{R}$ is received accumulatively, and computed as $r_{t_j} = \bar{r}_{t_j} + \hat{r}_{t_j}$, where $\bar{r}_{t_j}$ is a transition outcome average reward iteratively accumulated and $\hat{r}_{t_j}$ is a transition action expected reward iteratively accumulated. All rewards are in the interval $[0, 1]$.

For a transition outcome average reward $\bar{r}_{t_j}$ to be iteratively computed, we consider four types of transition outcome rewards as a set of rewards $\mathbb{R}_{to}$ including the r_{to_self}, $r_{to\_success}$, r_{to_back}, and r_{to_fail}.

- r_{to_self}: a self-transition reward for a successful transition that has $s_{origin} = s_{dest}$
- $r_{to\_success}$: a reward given to a successful transition that has $s_{origin} \neq s_{dest}$.
- r_{to_back}: the reward for a backtracked transition

- r_{to_fail}: the reward for a failed transition.

we compute the accumulated transition outcome average reward $\bar{r}_{t_j}$ for a transition t_j using Equation 5.1:

$$\bar{r}_{t_j} = \frac{1}{n_{t_j}} \sum_{i=1}^{n_{t_j}} r_{to_{(t_j,i)}} \tag{5.1}$$

where $r_{to_{(t_j,i)}} \in \mathbb{R}_{to}$ is a transition outcome reward received at the i'th iteration for a transition $t_j \in T$, and n_{t_j} is the number of times the transition t_j was selected.

The excepted transition action reward $\hat{r}_{t_j}$ is computed as the sum of the given rewards for pass/fail, weighted by how many times the two verdicts actually occurred. To compute $\hat{r}_{t_j}$ iteratively, we consider four different rewards for two types of transition actions (precondition and assertion) as a set of rewards $\mathbb{R}_{ta}$ including the $r_{precond\_pass}$, $r_{precond\_fail}$, r_{assert_pass}, and r_{assert_fail}.

- $r_{precond\_pass}$: the passed precondition reward
- $r_{precond\_fail}$: failed precondition reward
- r_{assert_pass}: passed assertion reward
- r_{assert_fail}: failed assertion reward

Then, the excepted transition action reward $\hat{r}_{t_j}$ for a transition t_j can be computed using Equations 5.2, 5.3 and 5.4:

$$\hat{r}_{t_j} = \hat{r}_{t_j\_precond} + \hat{r}_{t_j\_assert} \tag{5.2}$$

$$\hat{r}_{t_j\_precond} = \frac{C_{precond\_pass}}{C_{precond\_total}} \times r_{precond\_pass} + \frac{C_{precond\_fail}}{C_{precond\_total}} \times r_{precond\_fail} \tag{5.3}$$

$$\hat{r}_{t_j\_assert} = \frac{C_{assert\_pass}}{C_{assert\_total}} \times r_{assert\_pass} + \frac{C_{assert\_fail}}{C_{assert\_total}} \times r_{assert\_fail} \tag{5.4}$$

where, in Equation 5.2, $\hat{r}_{t_j\_precond}$ represents the expected precondition (action) reward for the transition t_j; $\hat{r}_{t_j\_assert}$ represents the expected assertion (action) reward for the transition t_j. Likewise, the *counts* for passed and failed preconditions and assertions, as well as their total number, are updated during each iteration as per Equations 5.3 and 5.4.

Then, our bandit heuristic search (BHS) strategy, proposed in [129], for test case generation becomes the following:

a) each transition $t \in T$ is selected once at the initialization of the strategy.

b) afterwards, the strategy iteratively select a transition $t_j \in T$ that maximizes

$$\bar{r}_{t_j} + \hat{r}_{t_j} + \sqrt{\frac{2 \ln n_{s_{origin}(t_j)}}{n_{t_j}}} \tag{5.5}$$

where $\bar{r}_{t_j}$ is the transition outcome average reward (in $[0, 1]$) for transition t_j, $\hat{r}_{t_j}$ is the transition action expected reward for transition t_j, n_{t_j} is the number of times transition t_j was selected, and $n_{s_{origin}(t_j)}$ is the number of times that the origin state s_{origin} of the transition t_j is visited and used to select transitions.

The overall steps for test case generation with our bandit heuristic search strategy are summarized in pseudocode in Algorithm 2 [129]. We have implemented this heuristic search strategy in Modbat. Modbat initializes a list of transitions `transitions` and an initial state s_0. The user need to initialize the number of test cases n, all counter variables (with 0 values), and reward variables. The function executeTransitions on Line 4 generates and executes a test case consisting of a sequence of selected transitions from an initial state s_0 to a terminal state $s_{terminal}$. On Line 6 in function executeTransitions, the function banditHeuristicSearch is called to select a transition `trans` using our bandit heuristic search strategy. Then, this selected transition `trans` is executed by the function executeTransition shown on Line 7, with a transition result of type `result` as the function return value. Meanwhile, function executeTransition calls the function updateExpectedReward on Line 16 to update the transition action expected reward $\hat{r}_{t_j}$ for the selected transition `trans` based on Equations 5.2, 5.3 and 5.4. After receiving the return value `result` on Line 7, the function updateAverageReward on Line 30 updates the transition outcome average reward $\bar{r}_{t_j}$ for `trans` with Equation 5.1, based on the result type of `trans`.

5.4.2 *Bandit Search-Based Test Suite Optimization*

The goal of the bandit heuristic search strategy is to guide the test case generation and obtain good results on the test adequacy criteria with smaller test suites containing less redundant test cases. The strategy relies on the configuration of eight different rewards to initialize the test case generation (as shown by the Require in Algorithm 2). Therefore, we need to find optimal solutions to configure these rewards and obtain optimized test adequacy criteria with considering their trade-offs.

5.4.2.1 *Test Adequacy Criteria as Multi Objectives*

For MBT, test adequacy criteria are often chosen to guide the automatic test case generation so that a good test suite is produced [125]. Modbat supports test adequacy criteria including state- and transition coverage [10] and linearly independent path coverage [128]. The state- and transition coverage indicates the number of states and transitions, respectively, that have been explored by a test suite. The linearly independent path coverage indicates the execution paths covered by a test suite. In addition to coverage, Modbat can also provide the measurement of failures found after a test suite is executed [10]. Thus, for our test case generation approach, we choose four different test adequate criteria:

- state coverage (Cov_s)
- transition coverage (Cov_t)
- linearly independent path (LIP) coverage (Cov_{lip})

Algorithm 2 Bandit Heuristic Search for Test Case Generation

Require: Initialize $s, transitions, n, r_{to\_self}, r_{to\_success}, r_{to\_back}, r_{to\_fail}, r_{precond\_pass}, r_{precond\_fail}, r_{assert\_pass}, r_{assert\_fail}, C_{precond\_pass}, C_{precond\_fail}, C_{assert\_pass}, C_{assert\_fail}$.

1: **func** executeTests
2: **for** $i = 1$ to n **do** ▷ n:number of test cases
3: executeTransitions

4: **func** executeTransitions
5: **while** s is not a $s_{terminal}$ **do** ▷ s:current state, starting from s_0
6: $trans \leftarrow$ banditHeuristicSearch($transitions$)
7: $result \leftarrow$ executeTransition($trans$)
8: updateAverageReward($result, trans$)

9: **func** banditHeuristicSearch($transitions$)
10: **if** $transitions$ has any never selected transitions **then**
11: **return** $t_{1st\_unselected}$ in $transitions$
12: **else**
13: **return** $t_{j'}$ in $transitions$ having $\text{argmax}\{\bar{r}_{t_j} + \hat{r}_{t_j} + \sqrt{\frac{2 \ln n_{s_{origin}(t_j)}}{n_{t_j}}}\}$

14: **func** executeTransition($trans$)
15: updateExpectedReward($trans$)

16: **func** updateExpectedReward($trans$)
17: **if** precondition of $trans$ **then**
18: **if** pass **then**
19: update $C_{precond\_pass} \mathrel{+}= 1$
20: **else**
21: update $C_{precond\_fail} \mathrel{+}= 1$
22: update $C_{precond\_total} = C_{precond\_pass} + C_{precond\_fail}$
23: **if** assertion of $trans$ **then**
24: **if** pass **then**
25: update $C_{assert\_pass} \mathrel{+}= 1$
26: **else**
27: update $C_{assert\_fail} \mathrel{+}= 1$
28: update $C_{assert\_total} = C_{assert\_pass} + C_{assert\_fail}$
29: update $\hat{r}_{trans} = \hat{r}_{t_{trans\_precond}} + \hat{r}_{t_{trans\_assert}}$ for $trans$ with Equations 5.3,5.4

30: **func** updateAverageReward($result, trans$)
31: **switch** $result$ **do**
32: **case** success
33: $r_{to_{(trans,i)}} = r_{to\_success}$
34: **case** self
35: $r_{to_{(trans,i)}} = r_{to\_self}$
36: **case** backtracked
37: $r_{to_{(trans,i)}} = r_{to\_back}$
38: **case** failed
39: $r_{to_{(trans,i)}} = r_{to\_fail}$
40: update $\bar{r}_{trans} = \frac{1}{n_{trans}} \sum_{i=1}^{n_{trans}} r_{to_{(trans,i)}}$ for $trans$

- the number of test cases used to find the first failure ($NTest_{fail_1}$)

We use these test adequacy criteria as objectives for multi-objective optimization.

5.4.2.2 *Multi-Objective Optimization*

We consider our bandit heuristic search problem as a multi-objective optimization problem. We tune the bandit heuristic search strategy shown in Algorithm 2 to find the optimal solutions for the reward parameter settings used in the test case generation. With the optimal solutions found to configure our bandit heuristic search strategy, we use them for the test case generation of Modbat models in general.

For this multi-objective optimization problem, we formally assume that a solution can be described in terms of an 8-dimensional reward decision vector $\vec{r}$ in the reward decision space $\mathcal{R}^8$. This solution can be used to initialize the generation of a test suite $ts \in TS$ (initialization of Algorithm 2), where TS is a set of test suites. Then, the vector-valued objective function $\vec{f} : \mathcal{R}^8 \to \mathcal{O}$ evaluates the quality of a specific solution by assigning it an objective vector $\vec{o} = \vec{f}(\vec{r})$ in the objective space $\mathcal{O}$. We define the reward decision vector as

$$\begin{aligned}\vec{r} = (&r_{to\_self}, r_{to\_success}, r_{to\_back}, r_{to\_fail},\\ &r_{precond\_pass}, r_{precond\_fail}, r_{assert\_pass}, r_{assert\_fail})\end{aligned} \tag{5.6}$$

and, according to our test adequacy criteria, we define the objective vector with four objectives as

$$\vec{o} = (f_1(\vec{r}), f_2(\vec{r}), f_3(\vec{r}), f_4(\vec{r})) = (Cov_s, Cov_t, Cov_{lip}, NTest_{fail1}). \tag{5.7}$$

We assume that all objectives are equally important and our goal is to optimize them. Therefore, to solve this multi-objective optimization problem, we need to find those reward decision vectors as solutions that optimize the vector-valued objective function $\vec{f} : \mathcal{R}^8 \to \mathcal{O}$. These solutions balance the trade-offs between the different objectives, and we measure the optimality of the solutions through the concepts of *Pareto optimality* and *Pareto dominance* [35] [43] [33].

Based on the concept of *Pareto dominance*, given two solutions $\vec{r} \in \mathcal{R}^8$ and $\vec{r}\,' \in \mathcal{R}^8$ as reward decision vectors which can be used to initialize two test suites ts and ts′, $\vec{r}$ is said to dominate, or *Pareto-dominate*, $\vec{r}\,'$ (written as $\vec{r} \succ \vec{r}\,'$) if and only if their objective vectors $\vec{o} = \vec{f}(\vec{r})$ and $\vec{o}\,' = \vec{f}(\vec{r}\,')$ satisfy: $\forall i \in \{1, 2, \ldots, k\}, \vec{f}(\vec{r}) \geqslant \vec{f}(\vec{r}\,') \wedge \exists i \in \{1, 2, \ldots, k\} : \vec{f}(\vec{r}) > \vec{f}(\vec{r}\,')$. We have $k = 4$ since we have four test adequacy criteria used as objectives. All reward decision vectors that are not dominated by any other reward decision vectors are said to form the *Pareto optimal set* $\mathcal{R}^{8*} \subseteq \mathcal{R}^8$, while the corresponding objective vectors are said to form the *Pareto frontier* $\mathcal{O}^* = \vec{f}(\mathcal{R}^{8*}) \subseteq \mathcal{O}$. Therefore, the *Pareto optimal set* $\mathcal{R}^{8*}$ contains only non-dominating reward decision vectors as optimal solutions to our multi-objective bandit search optimization problem. Each one of these vectors can then be used to initialize the generation of a test suite. Thus, we find an optimal subset of test suites $TS^* \subseteq TS$ which balance the trade-offs of our four different test adequacy criteria: no other subset of TS can improve one objective without making the other objectives worse.

In the article [129], we applied the jMetal Java-based framework [41, 101] for multi-objective optimization using metaheuristics to obtain the *Pareto optimal set* $\mathcal{R}^{8*}$ for our multi-objective bandit search problem. jMetal is specifically aimed at multi–objective optimization, and implements and supports a number of modern multi–objective optimization algorithms, such as the NSGA-II [39] algorithm, to obtain the *Pareto optimal set*.

Fig. 5.7 gives an overview of the implementation to solve our multi–objective optimization problem for Modbat models with the aid of jMetal v5 [101] and NSGA-II. The working principle of jMetal is based on algorithms, such as NSGA-II. Users need to first define their multi–objective optimization problem with an objective function, and then solve them with the chosen algorithm. We have implemented our multi–objective bandit search optimization problem in jMetal with our vector-valued objective function $\vec{f} : \mathcal{R}^8 \rightarrow \mathcal{O}$, and used the NSGA-II genetic algorithm provided by jMetal to solve this problem.

The process goes through generations of the NSGA-II algorithm with the number of evaluation rounds and population provided. For each generation, we run different Modbat models in parallel using 8 different values of the reward parameters (generated by the NSGA-II from jMetal) as input to our bandit heuristic search strategy. After the Modbat models have been executed, the results of the four test adequacy criteria for all models are used as the objective values and sent to jMetal so that they can then be used by the NSGA-II algorithm to generate reward values for the next generation. When all the generations of the NSGA-II algorithm are finished, jMetal provides two files containing the *Pareto optimal set* and the *Pareto frontier* found by the NSGA-II.

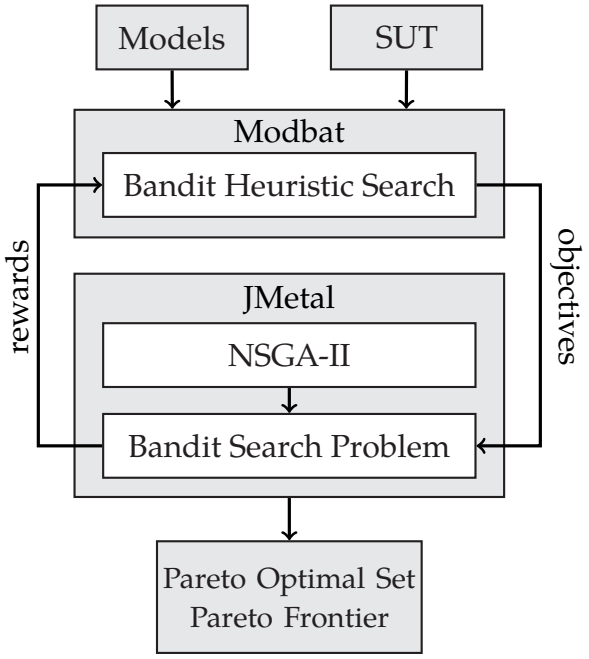


Fig. 5.7: Multi–objective bandit-search optimization.

5.5 Results and Contributions

The main contribution in article [128] is to propose an approach to capture and visualize test case execution paths of models. The information obtained by visualization, such as the number of linearly independent paths, can be used as test adequacy criteria to indicate the degree to which test paths have been executed. Our approach relies on first recording execution paths with a trie data structure, then visualizing them using state-based graphs (SGs) and path-based graphs (PGs) obtained by applying abstractions. To obtain the SGs and PGs, we have proposed abstractions as our initial technique to reduce the size and complexity of graphs.

We have evaluated our path coverage visualization approach on a collection of Modbat models. The results of the evaluation is shown in Table 5.1. The list of models includes the Java server socket implementation, the coordinator of a two-phase commit protocol, the Java array list and linked list implementation, and ZooKeeper. Among them, the array and linked list models, as well as the ZooKeeper model, consist of several parallel EFSMs, which are executed in an interleaving way. For each Modbat model, we have considered configurations with 10, 100, 200, 500 and 1000 randomly generated test cases.

Table 5.1 first lists the statistics directly reported by Modbat: the number of states (**S**) and transitions (**T**) covered for each model (together with their percentage), and the number of test cases (**TC**) and failed test cases (**FC**). The second part of the table provides the metrics of the SGs and PGs we generate. We list: the total number of **Nodes** (including both state nodes and choice nodes); the total numbers of **Edges** (**E**), the number of failed edges (**FE**), and loops (**L**). For the PGs, our path coverage visualizer also computes the numbers of linearly independent paths (**LIP**), the longest paths (**LP**), the shortest paths (**SP**), the average lengths of paths (**AVE**), and the corresponding standard deviation (**SD**).

When comparing the results of the SG and PG obtained from all the models shown in Table 5.1, we notice that the SG has a smaller number of nodes and edges than the PG for any increase in the number of test cases by going from 10 to 1000. This means that the SG is more abstract than the PG, and gives us an overview of how the models are executed. For the PG, we can directly see the information about the number of linearly independent paths (**LIP** column in Table 5.1), which tells us how execution paths are constructed and executed from the sequences of transitions executed.

The results in Table 5.1 also indicate the degree to which the models are executed by the generated test cases. Taking the coordinator model as an example, the numbes of nodes and edges in both the PG and SG do not increase after 100 test cases have been executed. Results like this give testers confidence about how well the models are explored by the tests. Also, for complex models such as the ZooKeeper model, the number of failed edges keeps increasing with more tests. This indicates that there might be parts that are hard to reach for complex models, so testers might need to increase the number or quality of the tests.

By comparing the results between SGs and PGs in Table 5.1, we notice, generally, that the SGs convey the behavior of the model well, while the PGs only show executed paths, without providing detail. A good path-coverage-based testing strategy requires that as many as possible the linearly independent paths are executed by the test cases.

Table 5.1: Experimental results for visualization of Modbat models.

Model	S	T	TC	FC	Path-based (PG)									State-based (SG)			
					Nodes	Edges			Paths					Nodes	Edges		
						E	FE	L	LIP	LP	SP	AVE	SD		E	FE	L
JavaNio ServerSocket	7/ 7 (100%)	17/17 (100%)	10	2	57	79	1	17	8	14	3	9.25	4.18	9	23	1	6
			100	3	177	243	1	48	30	15	2	7.87	3.84	9	23	1	6
			200	8	363	528	4	111	53	29	2	9.68	6.24	10	25	1	7
			500	14	779	1147	8	247	105	29	2	10.51	5.29	11	27	1	8
			1000	28	1269	1904	15	439	168	29	2	10.80	4.79	11	27	1	8
Coordinator Test	7/ 7 (100%)	6/ 6 (100%)	10	0	17	20	0	0	1	6	6	6.00	0.00	17	20	0	0
			100	0	21	27	0	0	1	6	6	6.00	0.00	21	27	0	0
			200	0	21	27	0	0	1	6	6	6.00	0.00	21	27	0	0
			500	0	21	27	0	0	1	6	6	6.00	0.00	21	27	0	0
			1000	0	21	27	0	0	1	6	6	6.00	0.00	21	27	0	0
ArrayList Iterator ListIterator	1/ 1 (100%) 2/ 2 (100%) 2/ 2 (100%)	11/11 (100%) 5/11 (45%) 12/29 (41%)	10	0	174	542	0	276	6	99	12	58.17	38.75	34	85	0	38
ArrayList Iterator ListIterator	1/ 1 (100%) 2/ 2 (100%) 2/ 2 (100%)	11/11 (100%) 9/11 (81%) 13/29 (44%)	100	0	1171	3222	0	1571	75	181	2	23.93	29.74	102	216	0	94
ArrayList Iterator ListIterator	1/ 1 (100%) 2/ 2 (100%) 2/ 2 (100%)	11/11 (100%) 10/11 (90%) 17/29 (58%)	200	1	3369	10474	1	4848	138	181	2	45.35	47.46	204	423	1	184
ArrayList Iterator ListIterator	1/ 1 (100%) 2/ 2 (100%) 2/ 2 (100%)	11/11 (100%) 10/11 (90%) 25/29 (86%)	500	1	10438	29730	1	14024	319	181	2	48.96	43.55	467	955	1	417
ArrayList Iterator ListIterator	1/ 1 (100%) 2/ 2 (100%) 2/ 2 (100%)	11/11 (100%) 10/11 (90%) 27/29 (93%)	1000	14	29056	86871	1	40609	649	406	2	70.87	64.17	896	1812	1	815
LinkedList Iterator ListIterator	1/ 1 (100%) 2/ 2 (100%) 1/ 2 (50%)	18/19 (94%) 8/11 (72%) 5/29 (17%)	10	0	216	718	0	348	9	191	10	56.11	72.01	34	85	0	36
LinkedList Iterator ListIterator	1/ 1 (100%) 2/ 2 (100%) 1/ 2 (50%)	19/19 (100%) 9/11 (81%) 7/29 (24%)	100	0	1190	3348	0	1679	83	191	2	23.51	38.05	148	312	0	131
LinkedList Iterator ListIterator	1/ 1 (100%) 2/ 2 (100%) 2/ 2 (100%)	19/19 (100%) 9/11 (81%) 19/29 (65%)	200	0	6266	17140	0	7549	178	191	2	54.45	49.14	405	824	0	295
LinkedList Iterator ListIterator	1/ 1 (100%) 2/ 2 (100%) 2/ 2 (100%)	19/19 (100%) 9/11 (81%) 22/29 (75%)	500	0	15091	43303	0	19797	406	257	2	60.17	61.56	699	1413	0	522
LinkedList Iterator ListIterator	1/ 1 (100%) 2/ 2 (100%) 2/ 2 (100%)	19/19 (100%) 9/11 (81%) 24/29 (82%)	1000	0	39391	113155	0	52461	825	257	2	74.66	67.19	1404	2819	0	1083
ZKServer ZKClient	4/ 4 (100%) 9/13 (69%)	4/ 4 (100%) 28/54 (51%)	10	0	488	536	0	6	10	27	17	24.60	2.65	158	203	0	6
ZKServer ZKClient	4/ 4 (100%) 11/13 (84%)	4/ 4 (100%) 38/54 (70%)	100	7	4628	5160	7	76	98	31	4	22.57	5.99	862	1110	5	75
ZKServer ZKClient	4/ 4 (100%) 11/13 (84%)	4/ 4 (100%) 39/54 (72%)	200	9	9869	9869	9	138	197	31	4	22.88	5.67	1532	1964	5	135
ZKServer ZKClient	4/ 4 (100%) 11/13 (84%)	4/ 4 (100%) 40/54 (74%)	500	26	27208	31918	25	325	480	31	4	22.79	5.31	3057	3910	10	320
ZKServer ZKClient	4/ 4 (100%) 11/13 (84%)	4/ 4 (100%) 43/54 (79%)	1000	47	63524	76090	44	648	937	31	4	23.01	5.07	5719	7201	16	643

We can directly see how many linearly independent paths are executed using the PGs. Even though the PGs might contain more nodes and edges as such, they are more scalable by avoiding crossing edges.

The overall result from our experimental evaluation on several model-based test suites shows that our abstraction technique reduces the complexity of graphs, and our visualization of execution paths helps to show the frequency of transitions taken by the executed paths and to distinguish successful from failed test cases.

The main contribution of article [129] is a heuristic search based test case generation approach for model-based testing, aiming at performing well on test adequacy criteria taking into account their trade-offs. We have proposed a bandit heuristic search strategy to handle the exploration versus exploitation dilemma for test case generation, and applied a multi-objective optimization technique to tune our strategy and optimize the chosen test adequacy criteria with the aid of the jMetal multi-objective optimization

framework and the NSGA-II Pareto-efficient algorithm. We have also evaluated our approach on a collection of Modbat models by comparing the results of the bandit-based heuristic search with a random test case generation approach. The models that we considered include four simple models used as a training set and two large and complex models used as an test set. The training set consists of the *ChooseTest* model [128], the *Java Server Socket* model [12], the *Java Array List* model [13], and the *Java Linked List* model [13]. The test set includes the *ZooKeeper* [11] and *PostgreSQL* [123] models.

Specifically, we have first applied our search strategy to the training set and used jMetal to optimize our search strategy. Then, the weights of the eight rewards in the resulting Pareto optimal set can be used as the optimal parameters for our strategy to the test set. All collected result data of the four test adequacy criteria are within 0 to 100; and are defined as follows:

- State coverage: $Cov_s \in \{0, \ldots, 100\}$
- Transition coverage: $Cov_t \in \{0, \ldots, 100\}$
- Score of Cov_{lip}: $Cov_{lip} * 2, Cov_{lip} \in \{1, \ldots, 50\}$
- Score of $NTest_{fail1}$: $102 - NTest_{fail1} * 2, NTest_{fail1} \in \{1, \ldots, 51\}$

We have calculated the two scores based on the fact that with 50 test cases, it is possible to have at most 50 linearly independent paths, and that the best possible outcome is that the first test finds a failure; but if no test finds a failure, we count the score as if the 51st test (which is never tried) would have found it.

In the article [129], since each parameter setting for 8 rewards was tested with 40 seeds and 50 test cases per seed on four models, we ran 8000 tests per parameter setting to determine fitness. We set up the jMetal framework with a population size of 50 and 100 generations for NSGA-II, so we ran a total number of 40 million tests in the training phase, which took four days using a 48 CPUs cluster. For the ZooKeeper model, we just applied 50 optimal solution candidates in the Pareto optimal set directly and collected the results for the test adequacy criteria; while for the PostgreSQL, we performed mutation testing using 86 mutants to inject 86 different errors to the PostgreSQL implementation. Then, we applied 50 optimal solution candidates to 86 mutated PostgreSQL, respectively.

We have visualized our result data using box plots. The result data are collected directly when jMetal finish all generations of the NSGA-II algorithm. Each box in the plot shows the range between the first and third quartiles (Q1 and Q3) as a rectangle. The solid red line in the box represents the median value. The distance between Q1 and Q3 is the inter-quartile range (IQR); 25% of the data lies below Q1 and 75% of the data lies below Q3. The blue dashed lines represent the smallest (largest) observed point from the dataset that falls within a distance of 1.5 times of the IQR below Q1 and above Q3, respectively. Circles represent outliers outside 1.5 times the IQR.

Fig. 5.8 and Fig. 5.9 visualize the box plots for two models from the training set: the Java Server Socket and Array List models and Rand represents the random approach; Heur represent the bandit heuristic approach. It can be observed that for the box plots of the Java Server Socket model, our bandit heuristic approach gives better results on the transition coverage Cov_t and the score of $NTest_{fail1}$ than the

random approach. However, concerning state coverage Cov_s and the score of Cov_{lip}, the random approach is slightly better. From the box plots of the Array List model in Fig. 5.8 and Fig. 5.9, we observe that our bandit heuristic approach achieves better performance on all objectives than the random approach. The box plots for both the Linked List and ChooseTest models show that the heuristic approach has better results on all objectives, which is similar to the results of Array List model. Hence, we do not specifically discuss their box plots here.

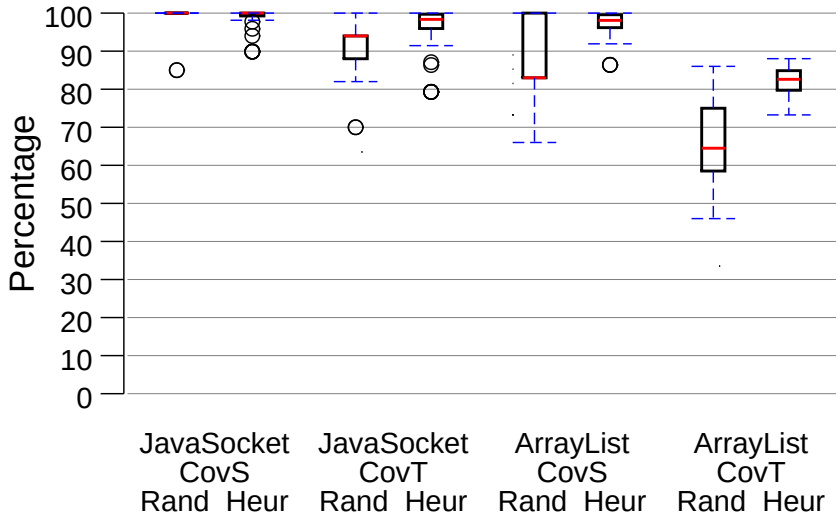


Fig. 5.8: Comparison of state and transition coverages for Java server socket and array list models

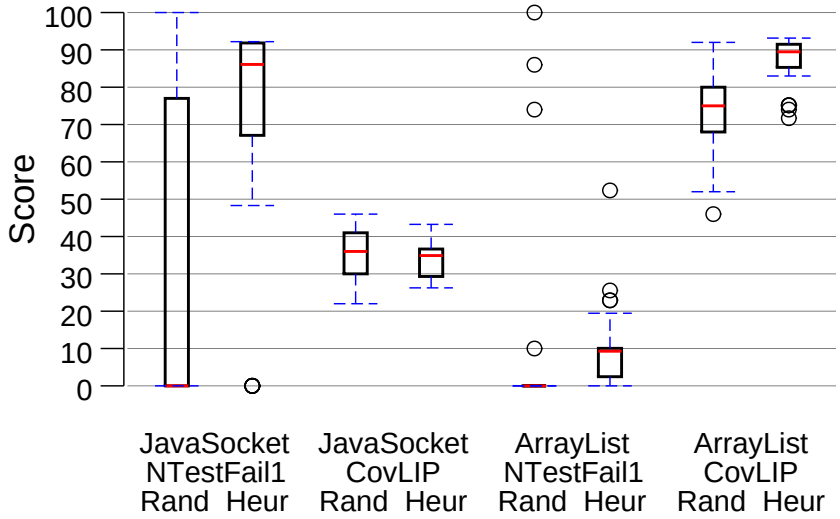


Fig. 5.9: Comparison of scores for number of test cases used to find the first failure and LIP coverage for Java server socket and array list models

After we obtained the Pareto optimal set from the training phase using the training set, we applied the resulting values of eight different rewards in the Pareto optimal set on the PostgreSQL and ZooKeeper models, respectively, from the test set. Fig. 5.10 and Fig. 5.11 show the box plots for the PostgreSQL and ZooKeeper models. We can see that, for the PostgreSQL model, the heuristic approach is slightly better than random

approach, since the box plot of the transition coverage Cov_t of the heuristic approach does not have the extra outlier (around 85) shown in the plot of the random approach in Fig. 5.10. For the resulting box plots of the ZooKeeper model in Fig. 5.10 and Fig. 5.11, the box plots of the random approach indicate better results on the four objectives than the heuristic approach. However, the box plots also show that the distribution of the result data for heuristic approach is more concentrated than the random approach. This can be observed from, for example, the box plot of the $NTest_{fail}$ score for the random approach. It has some extremely bad results (0) and extremely good results (100), compared to the box plot of the heuristic approach. The box plots of the heuristic approach from other models also reflect this characteristic, i. e., they have a more concentrated distribution of their resulting data than random approach.

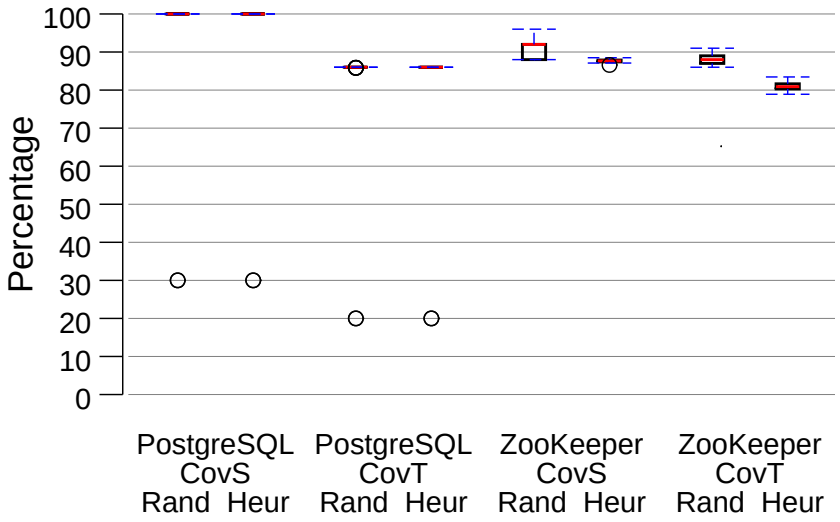


Fig. 5.10: Comparison of state and transition coverages for PostgreSQL and ZooKeeper models

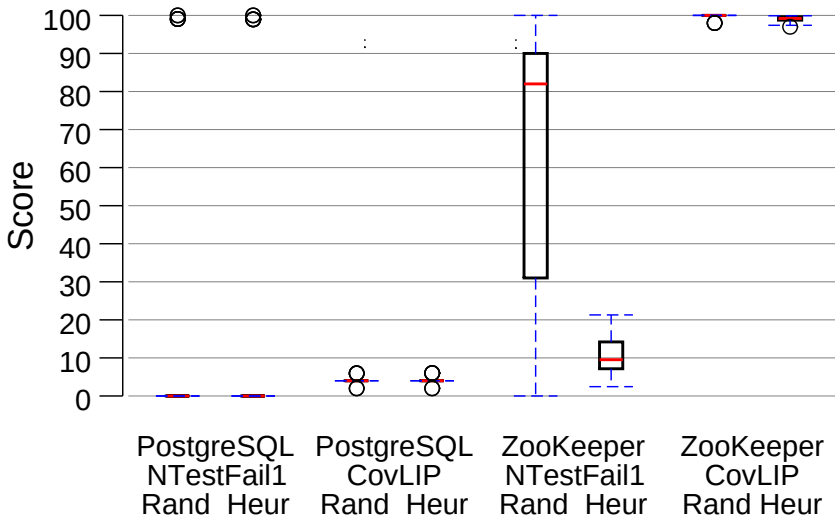


Fig. 5.11: Comparison of scores for number of test cases used to find the first failure and LIP coverage for PostgreSQL and ZooKeeper models

Our box plots have compared the performance between our bandit heuristic

approach and a random approaches for each objective, separately. However, our goal of applying multi-objective optimization to tune our bandit heuristic search strategy is to find optimal solutions balancing the trade-offs of our four different test adequacy criteria. This means that, to find out if the bandit heuristic approach has a potential to perform better, we also need to consider the trade-offs of the four objectives comprehensively. Therefore, in the article [129], we have computed the result of the global average for each model, shown in Table 5.2.

Table 5.2: Global averages obtained by heuristic and random approaches for each model.

Model	**Heuristic**		**Random**
	Max GA	**Aver GA**	**GA**
ChooseTest	76.81	61.11	50.80
JavaServerSocket	82.26	75.38	64.35
ArrayList	82.62	68.95	59.15
LinkedList	71.19	69.77	58.06
PostgreSQL	72.74	48.45	48.45
ZooKeeper	72.91	69.55	84.79

For all the models in the table, the column **GA** gives the global average for each model after applying the random approach. To obtain a global average result for each model, we first compute the average of collected result values for each objective obtained for the model using the 40 fixed seeds. Since we have four objectives (test adequacy criteria), we can obtain four average values. We then compute an average of these four average values as the global average **GA** to indicate an overall result of the random approach. The column **Aver GA** shows an average result of global averages computed for each model, after applying our heuristic approach. To obtain the result of **Aver GA** for each model, we first compute all the global averages (**GA**s) for all 50 candidates from the Pareto optimal set, then we compute an average over the 50 resulting global averages to get the result of **Aver GA**. **Max GA** is the maximum global average of the 50 resulting global averages computed and ranked for each model based on its result data. For the PostgreSQL model, we additionally average the global average over 86 mutants for both heuristic and random approaches.

From Table 5.2, we can observe that all models from the training set have better results by the bandit heuristic approach when comprehensively considering the trade-offs of the four objectives using global averages. This success on the training set indicates that our bandit heuristic search approach has the potential of being significantly better than a random search.

For the test set (PostgreSQL and ZooKeeper), the difference is less clear. The heuristic approach for PostgreSQL has better transition coverage, but the difference is not significant, and the average scores even match up to two digits after the decimal point. For ZooKeeper, we can see that the random approach performs better than the heuristic approach in Table 5.2. Thus, the results of the bandit heuristic search for the test set is not as good as the results from the training set. The reason for this is that our training set is too small (only four simple models), resulting in overfitting. Even so, the results of the PostgreSQL model from the test set show a potential for the bandit

heuristic search approach to perform much better than the random approach. This indicates that our bandit-based heuristic search approach has potential to obtain better and more predictable and consistent results on the chosen adequacy criteria compared to random test case generation, while considering the trade-offs of the test adequacy criteria.

An important contribution of the article [129] is an implementation of a search-based test case generation relying on our bandit-based heuristic search strategy as an extension for Modbat. This implementation has been included as a new feature in the Modbat 3.4 release. We have defined test adequacy criteria as multi-objectives so that Modbat implements our strategy for test case generation in addition to its standard random search. The reward parameter settings, obtained in the Pareto optimal set after applying NSGA-II provided by jMetal, can then be used to initialize the test case generation with our bandit-based heuristic search strategy to generate test cases for advanced Modbat models in general and targeting the chosen test adequacy criteria.

5.6 Related Work

In software testing, coverage analysis is an important concern. It can be used as a test adequacy criterion to decide whether additional test cases are needed, and related to which aspects of the SUT.

For the research related to the article [128], existing tools for source code coverage generally only report a verdict on which lines of code has been executed and how often. In the tool Tecrevis [74], redundancy in unit tests is visualized by providing a graphical mapping between each test case and the artifacts in the SUT (here: methods) that indicates which tests exercise the same component. In path coverage, the underlying graph is usually derived from the source code, the *control flow graph*, or from the *call graph* of the SUT, when considering function calls. In our approach, we are not directly concerned with visualizing paths of the SUT, but rather, paths on the testing model used for test case generation. Correspondingly, our graphs are usually more concise than the control flow graph, as not all branches of the SUT may need to be modeled at the level of ESFMs.

Coverage information is more understandable with the aid of visualization. Visualization of large state spaces is addressed by Ladenberger and Leuschel in the ProB tool [77]. In their approach, a coloring scheme used for states and transitions indicates whether the state space has been exhausted. However, they do not directly visualize coverage of the underlying model as in our approach. Moreover, they do not cover multiple transitions between the same pair of states as in our application scenario; however, this could be accounted for by adjusting the thickness of edges by the number of collapsed edges. Similarly, automated visualization is applied by Groote and van Ham [55] to the Very Large Transition System (VLTS) Benchmark set [38]. A relation between the graphical representation of the underlying model (in the form of UML sequence diagrams) and a set of paths from test cases is presented by Rountev et al. [113]. Their goal is to derive test cases, and as such they are not concerned with a representation of the paths.

The basic visualization elements of both SGs and PGs we have defined are based on the concept of *simple path* proposed by Ammann and Offutt [8]. That is, an execution

path is a simple path if there are no cycles in this path, with the exception that the first and last states may be identical (the entire path itself is a cycle) [8]. According to this definition, any execution path can be composed of simple paths. Therefore, we apply the concept of the simple path by considering only transitions from $s_{origin}(t)$ to $s_{dest}(t)$ (or $s_{origin}(t)$ if t is a self-transition or backtracked transition).

Several approaches related to our article [129] have been developed and documented in the literature, aiming at obtaining optimal test cases from test generation with multi-objective optimization and Pareto-efficient approaches. A technique for test data generation using multi-objective optimization was proposed by Oster and Saglietti [105]. It applied evolutionary algorithms to handle two objectives. One is the maximization of data flow coverage. The other is the minimization of the number of test cases required. The technique was used to test object-oriented programs implemented in Java with respect to control-flow and data-flow based criteria. Two different variants of genetic algorithms were used: the Multi-Objective Aggregation (MOA) and Non-dominated Sorting Genetic Algorithm (NSGA) in order to compare with a random approach and a simulated annealing algorithm. The experimental results showed that simulated annealing outperformed other algorithms. The authors also pointed out that NSGA offered a higher flexibility since it does not return a single optimal result as simulated annealing does, but a complete solution set instead.

Harman et al. [78] presented a multi-objective approach for search-based test data generation. The approach applied multi-objective optimization to optimize branch coverage and generate branch adequate test sets for branch adequate testing. The authors considered two objectives, including branch coverage and dynamic memory consumption, aiming at obtaining a good branch coverage while consuming as much dynamic memory as possible. The effectiveness of three search approaches were compared to generate branch adequate test data while maximizing dynamic memory addition. The approaches included a random search, a weighted genetic algorithm search, and a Pareto optimal search (NSGA-II). The experiment was carried out on testing C code from both real-world and synthetic examples.

For our approach, we defined four test adequacy criteria as objectives, including different coverage criteria and the failure detection, and considered their trade-offs. Instead of branch coverage, we considered linearly independent path coverage (LIP) as one of our test adequacy criteria, since path coverage is a stronger test adequacy criterion than branch coverage and it concerns a sequence of branch decisions instead of only one branch at a time. Also, our approach is based on models of the SUT. We optimized path coverage and other criteria at the model level rather than coverage of the SUT code. We considered that the process of test case generation faces the exploration versus exploitation dilemma, so we proposed the bandit heuristic search strategy to handle this dilemma and guide the test case generation. We also used a multi-objective optimization technique with the genetic algorithm NSGA-II to tune our strategy and optimize the four test adequacy criteria we defined. For the experimental evaluation, we also compared our approach with a random approach.

Our bandit-based heuristic approach also relates to some extent to work on search-based testing. Lei et al. [91] used random testing augmented with heuristics to find fault-revealing test cases more efficiently. In random testing, there exists the problem of choosing suitable input with the right values and types. These problems are taken care

of in model-based testing since the user provides a model that generates these inputs. Similarities exist in three of the six heuristics used in Guided Random Testing [91]: 1) *Impurity*: We use a different weight for self-loop transitions, which contains at least some impure methods as not to be completely redundant; 2) *Bloodhound*: We also choose transitions based on coverage, but at the model level instead of the SUT code; 3) *Orienteering*: At this point, we do not consider the time it takes to execute a transition, since in our examples the execution times of transition actions did not differ in major ways.

In addition to using Pareto-efficient approaches for search-based test generation, different fitness functions for white-box testing have been discussed by Salahirad et al. [115]. Their findings confirm that high (source code) coverage is a prerequisite for successful fault detection, and that branch coverage stands out as the most effective single criterion. They used *treatment learning* to discover which metrics best predicts fault detection. We have not investigated how different subsets of our criteria (especially when used within a limited resource budget) compare to each other, as we only have four, and hence much less than they had to consider. Rojas et al. [112] found that multi-objective optimization algorithms based on Pareto dominance are less suitable than a linear combination of the different non-conflicting objectives. It is not obvious how we would prioritize weights among the four different objectives, a question which also [112] left for their future work.

Related work also exists in multi-objective optimization for test selection. A multi-objective formulation of the regression test case selection problem is presented by Yoo and Harman [138]. They show how multiple objectives can be optimized using a Pareto efficient approach. They considered two criteria scenarios including a two-objective formulation combining code coverage and execution time, and a three-objective formulation combining code coverage, execution time, and the past fault detection history. The goal was to find a subset of a test suite which is a Pareto optimal set with respect to the chosen test criteria. Three algorithms were discussed to solve these two scenarios, including a reformulation of the single-objective greedy algorithm, NSGA-II, and vNSGA-II search-based approaches. The case studies were carried out on five programs in the Siemens suite and source code from the European Space Agency. For each program of the case studies, four test suites were randomly selected from existing available test suites, and then used as input to the multi-objective Pareto optimization process. The results showed that the search-based approaches can out-perform the greedy approach.

Multi-objective test case selection techniques were studied by Mondal et al. [98] to analyze both coverage-based and diversity-based test case selection. A novel approach was presented for bi-objective optimization scenarios to maximize code coverage/test case diversity and to minimize test execution time. The authors also proposed a three-objective optimization approach that maximizes both code coverage and test suite diversity at the same time, while minimizing the test execution time. The results of two- and three-objective optimization were compared in terms of fault detection rate, on 16 versions of five real-world programs, such as JBoss and Apache Ant. The authors used the Additional-Greedy and NSGA-II algorithms for bi-objective optimization, but only applied NSGA-II for three-objective optimization. The results of their experiment showed an improvement of the fault detection rate by the three-objective optimization

approach. For our approach, we do not have existing available test suites, so we focus on using the bandit-based heuristic search to generate optimal test cases directly, with the aid of the Pareto-efficient approach to optimize the four test adequacy criteria. Also, we did not consider test execution time as an objective, but we used the number of test cases to find the first failure in conjunction with three different coverages as objectives.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

In this chapter, we first summarize the thesis by re-visiting the research questions and discussing the main contributions, and then outline directions for future work.

6.1 Research Questions Revisited

The main research focus of this thesis was on model-based testing (MBT) for the development and testing of distributed systems and protocols. In this section, we re-visit our research questions and summarize our answers to these questions in the context of the major activities of MBT as shown in Fig. 6.1.

RQ1: *How can model-based testing be applied to detect errors and to ensure the correctness of quorum-based fault-tolerant distributed systems and protocols?*

RQ2: *How can Coloured Petri Nets and CPN Tools be used to support model-based testing for distributed systems and protocols?*

RQ3: *How can test criteria and test case generation technology of model-based testing measure test adequacy and effectively generate test cases?*

RQ1 motivated the investigation of MBT approaches and related testing artifacts used during the MBT process, as shown in Fig. 6.1, for quorum-based distributed systems and protocols in general. We have addressed this research question via the Gorums framework and CPNs, and this research question resulted in two journal articles [132, 133]. The goal of this research question was to provide an MBT approach to validate the correctness of quorum-based fault-tolerant distributed systems implemented with the Gorums framework, and to validate the correctness of the Gorums library.

We have developed an MBT approach captured by our QuoMBT framework based on CPNs and test case generation from CPN models to test quorum-based fault-tolerant distributed systems and protocols implemented with the Gorums framework. CPNs was chosen as the foundation for our MBT approach and the QuoMBT framework, since CPNs enables compact modeling of data and data manipulation which is required for message modeling, quorum functions modeling, and concrete test case generation. Also, CPN Tools can be used to perform model validation prior to test case generation.

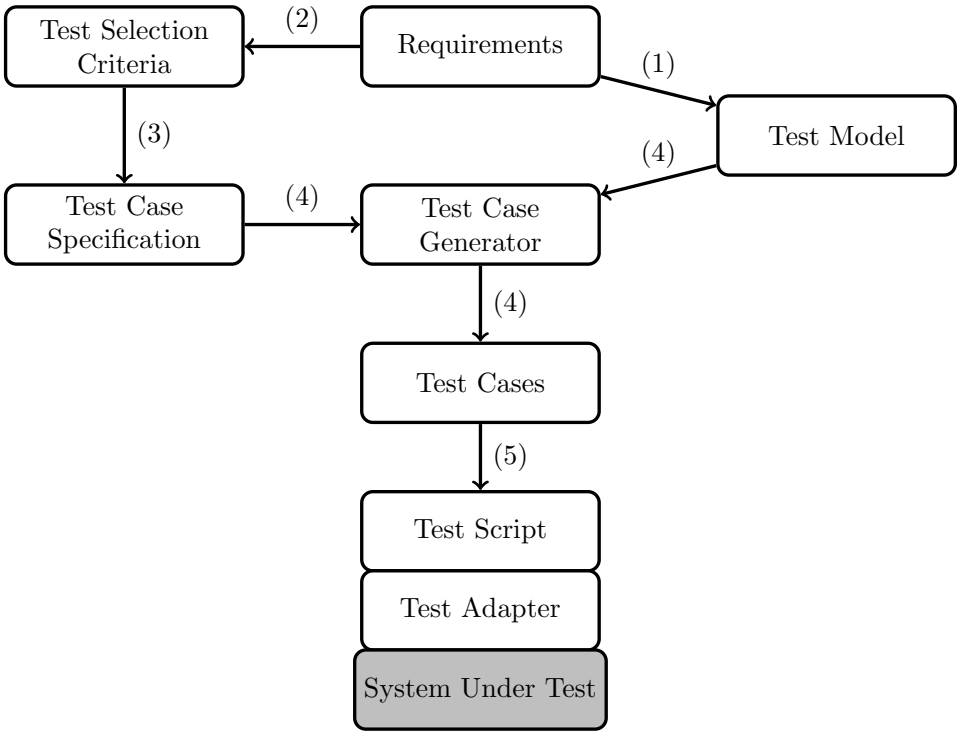


Fig. 6.1: The process of model-based testing [124].

The mature support of the CPN Tools for both simulation and state exploration has been important in our approach in order to facilitate practical experiments. This has been important to enable our MBT approach and conduct practical experiments and evaluation of MBT applied to quorum-based distributed systems and protocols.

We have constructed CPN models in order to evaluate our MBT approach and the QuoMBT framework, and to test the implementations of the single-writer and multi-reader distributed storage system and the single decree Paxos protocol implemented with the Gorums framework. The Gorums framework can help to simplify the main quorum logic and control flow of implementations. However, it was equally important to test the Gorums middleware implementation with our MBT approach in order to ensure its correctness. Our MBT approach and the QuoMBT framework cover most of the activities from (1) to (5) of the MBT process in Fig. 6.1, involving modeling patterns, test case generation algorithms, and a test case execution infrastructure. The testing artifacts of our MBT approach cover test cases, test inputs, test outputs, test oracles, test scripts, test adapters, and test drivers.

RQ2 led to the goal of developing software tools and techniques based on CPNs and CPN Tools in order to support our MBT approach for testing distributed systems and protocols. Software tools and techniques can help to detect errors and ensure stable operation of the software systems. For this research question, we have mainly focused on the development of software engineering testing tools to support MBT involving the *test case specification* and *test case generation* activities in Fig. 6.1.

We have implemented the MBT/CPN tool as an extension for the CPN Tools and presented the tool in article [134]. Although CPNs and CPN Tools have been widely used for modeling, validation, and verification of concurrent software systems, their application for MBT had only been explored to a limited extent prior to this thesis. The

MBT/CPN tool supports both simulation- and state space-based test case generation from CPN models. The main idea underlying the MBT/CPN tool is for the modeler to capture the observable input and output events (transitions) in a test case specification. We have applied the MBT/CPN tool supporting our MBT approach and the QuoMBT framework to validate the implementation of the Gorums middleware framework and to test the single-writer and multi-reader distributed storage system and the single decree Paxos protocol implemented via Gorums. The MBT/CPN tool has also been used to test the implementations of a two-phase commit transaction (2PC) protocol implemented without using Gorums to show its broader application for MBT.

RQ3 focused on the *test selection criteria* and *test case generation* activities in Fig. 6.1, aiming at finding better approaches for measuring test adequacy and effectively generating test cases in order to obtain good test cases and desired results in MBT. We have addressed this research question via the Modbat tester. The reason to choose the Modbat tester as our foundation of this research question instead of CPNs was that Modbat has a software architecture and existing benchmark suites that enabled us to further develop test criteria and test case generation techniques. The articles [128, 129] have addressed this research question.

For MBT, it is a challenge to generate sufficiently many and diverse test cases for a good coverage of the model and the SUT. Especially for complex software systems, the decision of how many tests to generate is challenging. For this reason, we have proposed an approach to capture and visualize execution path coverage of test cases on the model level as a test adequacy criterion in article [128]. Our approach relies on the use of a trie data structure to capture execution paths of test cases and proposed visualization abstractions as the foundation for path coverage visualization. The visualization of execution path coverage can provide a better overview of how the different parts of the model have been explored, if the tests have redundancies, and if any parts of the system are hard to reach via the test case generation.

MBT can automatically generate test cases from abstract (formal) models of the SUT. However, the random search approach for test case generation might result in test suites having redundant test cases which only cover few distinct execution paths of the models and the SUT. This means that with a random approach, it is a challenge to decide how many and diverse test cases to generate so that we can obtain desired results on test adequacy criteria such as path coverage. Prior to this thesis, the Modbat tester only had its standard random search approach to generate test cases. To tackle these challenges, we have considered that the test case generation faces the exploration versus exploitation dilemma, and that obtaining desired and balanced results on test adequacy criteria is a multi-objective optimization problem. We have proposed a search based test case generation approach in article [129] for the Modbat tester. The approach relies on a bandit heuristic search strategy that we have implemented as an extension to the Modbat tester and a multi-objective optimization technique. It is aimed at finding and generating a subset of test cases that optimizes the results of the chosen test adequacy criteria. The bandit heuristic search strategy can address the exploration versus exploitation dilemma and guide the test case generation. We have defined four test adequacy criteria as multi-objectives and applied a multi-objective optimization technique based on the jMetal framework and the NSGA-II genetic algorithm to tune our strategy to get optimal test suites and to balance the chosen criteria.

6.2 Summary of Contributions

Our research method on MBT for distributed systems and protocols lead to contributions in three main areas, shown as ellipses in Fig. 6.2. In this section, we summarize our contributions for each main area, by first presenting our contributions to the *Theoretical Foundations and Approaches*, followed by the contributions to the *MBT Software Tools and Techniques*, and then the contributions to the *SUT Case Studies and Experiments*. As can be seen from Fig. 6.2, these three areas are closely connected to each other by our research method and activities.

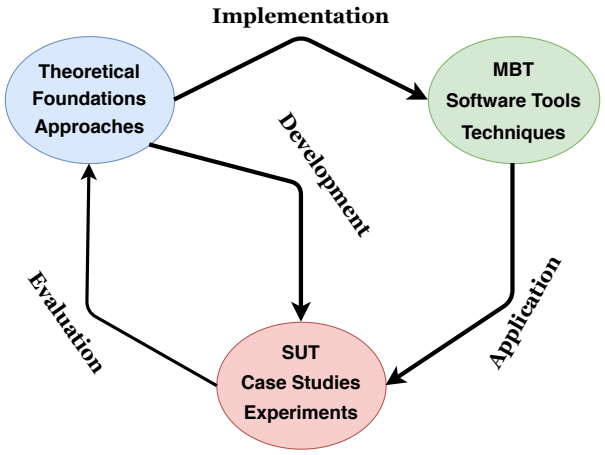


Fig. 6.2: Research method and activities.

6.2.1 *Contributions to the theoretical foundations and approaches*

The first contribution in this area is the MBT approach captured by the QuoMBT framework developed based on formal modeling and test case generation using CPNs to test quorum-based fault-tolerant distributed systems and protocols implemented via the Gorums framework.

An important attribute of our approach is that the CPN models have been constructed such that they can serve as a basis for MBT of other quorum-based distributed systems and protocols implemented with the abstractions of the Gorums framework. In other words, given a distributed system implemented by the Gorums framework, it is only the implementation of the quorum functions that needs to be changed when modeling the behaviors of quorum calls and quorum functions. The use of this modeling patterns ensure the generality of our MBT approach with CPNs. Our MBT approach and the QuoMBT framework can perform both unit tests and system level tests under both common successful execution scenarios and failure scenarios involving server crashes and injected programming errors to obtain high code coverage. This has been successfully proved by our case studies which include a single-writer and multi-reader distributed storage system and a single-decree Paxos consensus protocol both implemented with the Gorums middleware.

The second contribution in this area is the approach for the Modbat tester to use a trie data structure to capture execution paths of test cases of models and visualize

them using graph abstractions. The approach focuses on execution path coverage as a test adequacy criterion to be visualized.

Our approach provides two types of visualizations for execution path coverage, the state-based graphs (SGs) and path-based graphs (PGs). The abstractions we have developed to simplify these graphs enable us to deal with the complexity of moderately large systems. Our visualization approach also relies on the attributes of graphs including the edge thickness to visualize the frequency of transitions on executed paths and edge colors of the graphs to show what kinds of tests succeed or fail. This means that our visualization of execution path coverage helps in obtaining an overview as to whether different parts of the model have been explored equally well and the frequency of transitions taken by the executed paths.

The third contribution is the search-based test case generation approach that we proposed based on the bandit heuristic search strategy we implemented in Modbat and optimized using a multi-objective optimization technique. The approach aims to generate a subset of test cases which optimize and balance the trade-offs of four chosen test adequacy criteria, including the path coverage as proposed in article [128].

Our bandit heuristic search strategy guides test case generation by addressing the exploration versus exploitation dilemma, and our approach optimizes the bandit heuristic search strategy through the jMetal multi-objective optimization framework and the NSGA-II Pareto-efficient algorithm. In this way, we obtain optimal test suites based on the Pareto optimal set provided by NSGA-II. Our search-based test case generation approach has been demonstrated by the case study on a collection of Modbat models, including a training set and a test set involving, e. g., the ZooKeeper distributed coordination service and the PostgreSQL database system.

6.2.2 Contributions to the MBT software tools and techniques

The tools that we have developed mainly focus on test case generation and test adequacy criteria. We have presented the MBT/CPN tool developed for CPN Tools to perform automated test case generation from CPN models. A main facility of the MBT/CPN tool is the uniform support for both state space and simulation-based test case generation. Our MBT/CPN tool has been effectively used to support our MBT approach and the QuoMBT framework. We successfully applied the MBT/CPN tool to test a distributed storage system and a Paxos consensus protocol both implemented via the Gorums framework. The general use of the tool has been demonstrated by validating the correctness of a two-phase commit transaction protocol. The demonstrations have shown that the MBT/CPN tool can successfully support our MBT approach and test cases generated by the MBT/CPN tool yield high statement coverage and detect errors in the implementations.

The second tool is the path coverage visualizer for Modbat. It has been developed based on the Modbat tester and enables the visualization of path coverage with our proposed abstractions for execution paths of test cases. The visualizer uses the abstractions to visualize execution paths based on test cases in two types of simplified visualizations: state-based graphs (SGs) and path-based graphs (PGs). With the aid of our abstractions, it is possible to visualize execution path coverage of moderately large and complex system models. The tool has been used to visualize execution path

coverage of test cases of a collection of examples, such as the ZooKeeper distributed coordination service.

The third tool developed as part of this thesis is for test case generation with the Modbat tester. We implemented the bandit heuristic search strategy as a search-based test case generation tool extension for the Modbat tester. Users of the tool can provide an initial configuration of reward values to set up the bandit heuristic search strategy in order to guide the test case generation while considering the trade-offs of exploration versus exploitation of states/transitions. We have defined test adequacy criteria as multi-objectives so that Modbat implements our strategy for test case generation in addition to its standard random search. We have also demonstrated how we can apply multi-objective optimization to tune our strategy to generate better test cases in general and target the chosen test adequacy criteria. Both tools for Modbat have been included as new features of the Modbat 3.4 release.

6.2.3 Contributions to the SUT case studies and experiments

In this thesis, we have performed two main case studies to evaluate our MBT approach and the QuoMBT framework. They involved a single-writer and multi-reader distributed storage system and a single decree Paxos protocol as the SUT, both implemented with the Gorums framework. We have performed MBT to test these two systems using our proposed MBT approach and the QuoMBT framework by constructing CPN models with CPN Tools and generating test cases with our MBT/CPN tool for the SUT. The experimental evaluation of the two case studies has indicated that we can obtain high code coverage of the SUT. Results for the distributed storage system show that we have obtained 100% code coverage for the quorum functions (unit tests), 96.7% statement coverage on the quorum calls, and 52.3% coverage on the Gorums framework (system tests). Results for the Paxos consensus protocol show that the statement coverage of unit tests are up to 90%. For the system tests of the Paxos implementation, the statement coverage for the quorum calls reaches 83.9%; the results of statement coverage for other Paxos core components are similarly high; for the Gorums library as a whole, the highest statement coverage reaches 51.8%. These results indicate that our MBT approach and the QuoMBT framework can successfully obtain good code coverage of the implementations.

In addition, for both case studies, our MBT approach have detected errors in the implementations. For the distributed storage implementation, we have detected injected errors and an error in a particular code path involving passing a *nil* message to either the read quorum call or write quorum call, which is not handled by a feature used to address this *nil* message in Gorums. We have reported this specific error to the developers of the Gorums framework. For the Paxos implementation, we found programming errors that included the leader detector electing a wrong leader; only the leader's failure detector being executed; the elected new leader obtaining a wrong round number; clients being unable to receive responses from the Paxos replicas; the Paxos system only being able to handle one request from one client; after the current leader fails, the failed leader executing the Paxos phases again. These testing results have all shown that our MBT approach is able to detect non-trivial programming errors in the implementations of complex distributed protocols.

Another case study was developed to test a coordinator implementation of a two-phase commit transaction (2PC) protocol using the MBT/CPN tool to generate test cases. This demonstrated the uniform support for both simulation and state space-based approaches and it demonstrated the general use of the MBT/CPN tool. The experimental results showed that we obtained a high statement coverage (94.7%) with test cases generated by both state space and simulation-based approaches. The results also showed that the statement coverage for the simulation based approach is not as good as for the state space based approach in some scenarios. The reason is that the simulation based approach generally cannot cover all the possible executions of the CPN model in the absence of a guided search for test case generation. However, in some scenarios, the state space based approach without guided search may suffer from the state space explosion problem.

Two case studies related to the Modbat have been carried out for evaluation purposes. We have performed experiments using our approach for execution path coverage visualization to capture execution paths and visualize their path coverage in SGs an PGs on a collection of Modbat models, including the ZooKeeper distributed coordination service. We have compared the experimental results of the SGs and PGs obtained from all the models. The results showed that the SGs are good at giving an overview of the models based on extended finite state machines with detailed information about executed states and transitions; the PGs show distinct executed paths well, with a particular focus on giving information on linearly independent path coverage which can be considered as a test adequacy criterion. A good path coverage-based testing strategy requires that test cases can execute as many linearly independent paths as possible. Therefore, the visualization of linearly independent paths can be used as a test adequacy criterion to indicate the degree to which test paths have been executed. Also, the results of SGs and PGs have indicated the degree to which the models are executed and covered by the generated test cases. This is done by our graphs using edge thickness to visualize the frequency of transitions on executed paths and what kinds of paths have higher coverage than others, and using different edge colors to show what kinds of tests succeed or fail. All these visual feedbacks show that our approach gives testers confidence about how well the models are explored by the generated tests.

The second case study involving the Modbat tester was to apply our search-based test case generation approach for a collection of Modbat models from both a training set and a test set, including models such as the ZooKeeper distributed coordination service and the PostgreSQL database system. Our search-based test case generation approach is based on 1) our bandit heuristic search strategy implemented for Modbat to guide test case generation; and 2) applying the jMetal multi-objective optimization framework with the NSGA-II Pareto-efficient algorithm to obtain optimal test suits. The experimental evaluation compared our bandit heuristic search strategy with the random approach for test case generation of models from both a training set and a test set. The result showed that test cases generated using the search-based approach of our bandit heuristic search strategy obtain more predictable and better state- and transition coverage, find failures earlier, and provide improved path coverage, while balancing the trade-offs of these test adequacy criteria. Additionally, the reward parameter settings in the Pareto optimal set obtained from the NSGA-II algorithm can be used to initialize

the search-based test case generation with our bandit-based heuristic search strategy to generate test cases for advanced Modbat models in general and targeting the chosen test adequacy criteria.

6.3 Future Work

The work presented in this thesis provides several directions for future work into model-based software testing for distributed systems and protocols. In this section, we discuss future work based on our research method and activities, by outlining research directions for future work in the ares of *theoretical foundations and approaches*, *MBT software tools and techniques*, and *SUT case studies and experiments*.

6.3.1 Theoretical foundations and approaches

First, we have studied MBT with CPNs for quorum-based fault-tolerant distributed systems and protocols implemented with Gorums framework. With our proposed MBT approach, we have obtained good coverage results on the quorum functions of unit tests and quorum calls of system tests for both a distributed storage system and a Paxos distributed protocol, which has considered both successful and failure scenarios. In order to increase coverage and consider more of the Gorums library's code paths, we need to test the distributed system and protocols implemented with Gorums under additional failure scenarios and adverse conditions, such as network errors and partitions. This will require finding an approach to specify and configure such failure scenarios in the CPN testing models. This will also require an extension of the test adapters such that they can control these failure scenarios and execute the distributed systems and protocols under test with additional parameter settings in the test cases.

Another future direction is to extend our MBT approach to be applicable also to non-quorum-based distributed systems. Here, it becomes important to investigate in more detail the test coverage obtained using simulation versus the test coverage obtained with state space exploration of CPNs, and how we can augment these two approaches with a guided search approach for test case generation. We anticipate that this will motivate work into approaches for on-the-fly test case generation and test case selection during state space exploration with CPNs, such as our bandit heuristic search strategy in the Modbat tester. This will lead to the question of how we can connect on-the-fly test case generation to the system under test with the test adapter.

For our approach to path coverage visualization, future work involves applying additional abstractions to further reduce the complexity of larger graphs, further developing our visualization approach to support more visualization features in the SGs and PGs, and extend it to other coverage criteria such as branch-coverage of boolean subexpressions within preconditions and assertions. The more detailed *modified condition/decision coverage* (MC/DC) could be used to refine the intermediate execution steps even further. In essence, many of the coverage techniques available at the SUT-level could be lifted to the model level to achieve visualization. Our approach for path coverage visualization can also be extended to support measuring path coverage not only at the model-level, but also at the SUT-level.

Future research directions also involve improving our test case generation approach with the bandit heuristic search and multi-objective optimization. Although the results of our heuristic search from the training set are promising, the results from the test set should be improved, especially for large Modbat models such as the ZooKeeper model. To address this problem, we need to develop more diverse Modbat models for the training set, then perform multi-objective optimization on them, and get well-fitted reward parameter settings in the Pareto optimal set. This will enable us to leverage the full potential of our approach. Also, we expect that it is possible to obtain better results in the Pareto optimal set by increasing the size of the population and the number of generations for the NSGA-II algorithm. Alternative algorithms provided by the jMetal framework can be evaluated to solve our multi-objective optimization problem. In addition, other test adequacy criteria may be considered as objectives for the optimization, such as the execution time of test cases. In addition to a bandit heuristic search strategy, further heuristic search strategies for test case generation could be explored. For another research direction, it is worth investigating self-optimizing approaches at run-time to further exploit the potential of our bandit heuristic search strategy for test case generation.

6.3.2 MBT software tools and techniques

Several directions are also interesting to pursue further related to the development of the MBT/CPN tool. Currently, the users of the tool need to manually implement the methods provided by the interface in the tester of the test adapter generated by test execution engine in order to use these methods to interact with the SUT. Therefore, one area of future work is to provide a higher degree of automation and support online testing for the MBT/CPN tool so that it can directly connect to a generated test adapter. This would make it possible to perform fully automated MBT to the SUT. Another direction for the MBT/CPN tool is to implement heuristic search strategy for the simulation-based and state space-based test case generation with CPNs, such as the bandit heuristic search implemented for Modbat, to guide test case generation and obtain optimal test cases.

For our path coverage visualizer tool for Modbat, we may investigate additional abstractions to support larger and more complex Modbat models, further reduce redundancy in the graphs, and support visualization not only for the Modbat models but also for the SUT. It is also interesting to investigate how to integrate our path coverage visualizer tool into other test platforms such as CPN Tools. The path coverage visualizer tool can also be extended to a test adequacy criteria tool to visualize different kinds of coverage such as statement coverage, branch coverage, or MC/DC.

For search based test case generation, it relevant implementing other smart strategies as tool extensions for Modbat or CPN Tools. This requires, for example, the investigation of other machine learning algorithms for state space search and planning, so that we can obtain less redundant test cases and desired testing results that balance the test adequacy criteria.

6.3.3 Case studies and experiments

To further evaluate the applicability of our MBT approach to fault-tolerant distributed systems and protocols, especially under additional failure scenarios, we need to apply our approach on additional quorum-based distributed systems and protocols. Therefore, in addition to the current case studies, it is highly relevant to evaluate our testing approach to additional complex distributed systems and protocols under failure scenarios such as network partition and recovery.

One way to achieve this is to extend the current distributed storage to support multi-writer storages with multiple clients and extend single-decree Paxos to multi-decree Paxos, and then investigate how these systems and protocols can address network partition and recovery and verify their correctness by applying MBT. This will also challenge the limits of state space-based generation of test cases.

For path coverage visualization and search-based test case generation with the Modbat tester, it will be important to develop more diverse Modbat models to further advance the applicability of our visualization approach and test case generation approach.

BIBLIOGRAPHY

[1] Jepsen:Distributed Systems Safety Analysis. https://jepsen.io. 1

[2] The Modbat 3.4. https://github.com/cyrille-artho/modbat/tree/3.4. 5

[3] Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, Dec 1990. 1.2.2

[4] PhD Thesis Work Github Repository. https://github.com/wruiwr/PhDThesisWork, April 2020. 1.9

[5] M. A. Adamski, A. Karatkevich, and M. Wegrzyn. *Design of embedded control systems*, volume 267. Springer, 2005. 1.5

[6] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Distributed computing*, 13(2):99–125, 2000. 2.3.1, 2.3.3

[7] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed computing*, 2(3):117–126, 1987. 2.6

[8] P. Ammann and J. Offutt. *Introduction to software testing*. Cambridge University Press, 2016. 1.2.3, 5.6

[9] Apache Software Foundation. Apache Cassandra. http://cassandra.apache.org. 3.6

[10] C. Artho, A. Biere, M. Hagiya, E. Platon, M. Seidl, Y. Tanabe, and M. Yamamoto. Modbat: A model-based API tester for event-driven systems. In *Haifa Verification Conference*, volume 8244 of *Lecture Notes in Computer Science*, pages 112–128. Springer, 2013. 1.4, 5.4.2.1

[11] C. Artho, Q. Gros, G. Rousset, K. Banzai, L. Ma, T. Kitamura, M. Hagiya, Y. Tanabe, and M. Yamamoto. Model-Based API Testing of Apache ZooKeeper. In *2017 IEEE Intl. Conf. on Software Testing, Verification and Validation (ICST)*, pages 288–298, March 2017. 1.4, 3.6, 5.5

[12] C. Artho and G. Rousset. Model-based testing of the Java network API. *arXiv preprint arXiv:1703.07034*, 2017. 5.5

[13] C. Artho, M. Seidl, Q. Gros, E. Choi, T. Kitamura, A. Mori, R. Ramler, and Y. Yamagata. Model-based testing of stateful APIs with Modbat. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 858–863, Nov 2015. 5.5

[14] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing Memory Robustly in Message-passing Systems. *J. ACM*, 42(1):124–142, Jan. 1995. 1.1.2, 1.1.3

[15] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002. 5.4.1

[16] C. Baier and J.-P. Katoen. *Principles of model checking*. MIT press, 2008. 1.3.2

[17] P. Berman, J. A. Garay, and K. J. Perry. Towards optimal distributed consensus. In *30th Annual Symposium on Foundations of Computer Science*, pages 410–415. IEEE, 1989. 2.5.3

[18] D. A. Berry and B. Fristedt. Bandit problems: sequential allocation of experiments (monographs on statistics and applied probability). *London: Chapman and Hall*, 5:71–87, 1985. 5.4.1

[19] J. Billington, G. E. Gallasch, and B. Han. A coloured petri net approach to protocol verification. In *Advanced Course on Petri Nets*, pages 210–290. Springer, 2003. 1.5

[20] M. Billington, J. Diaz, and G. Rozenberg. Application of petri nets to communication networks, advances in petri nets, volume 1605 of lncs, 1999. 1.5

[21] M. Brambilla, J. Cabot, and M. Wimmer. Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering*, 1(1):1–182, 2012. 1

[22] J. Brito and A. Castillo. *Bitcoin: A primer for policymakers*. Mercatus Center at George Mason University, 2013. 2.5.3

[23] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012. 5.4.1

[24] M. Burrows. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *Proc. of the 7th Symp. on Operating Systems Design and Implementation*, OSDI '06, pages 335–350. USENIX Association, 2006. 2.5.3

[25] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011. 2.1, 2.2, 2.3, 2.3.2, 2.5.1

[26] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *Proc. of the Twenty-sixth Annual ACM Symp. on Principles of Distributed Computing*, PODC '07, pages 398–407. ACM, 2007. 3.6

[27] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43:225–267, March 1996. 2.3.3

[28] K. Cheng and A. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *Proc. 30th Intl. Design Automation Conf.*, DAC, pages 86–91, Dallas, USA, 1993. ACM. 1.4

[29] K. Cheng and A. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *Proc. 30th Intl. Design Automation Conf.*, DAC, pages 86–91, Dallas, USA, 1993. ACM. 1

[30] K.-T. Cheng and A. S. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *30th ACM/IEEE Design Automation Conference*, pages 86–91. IEEE, 1993. 5.1

[31] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, 1994. 1.2.4

[32] G. Chockler, R. Guerraoui, I. Keidar, and M. Vukolic. Reliable distributed storage. *IEEE Computer*, 2008. 1.1.2

[33] C. A. C. Coello, G. B. Lamont, D. A. Van Veldhuizen, et al. *Evolutionary algorithms for solving multi-objective problems*, volume 5. Springer, 2007. 5.4.2.2

[34] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG System: An Approach to Testing based on Combinatorial Design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997. 4.4

[35] Y. Collette and P. Siarry. *Multiobjective optimization: principles and case studies*. Springer Science & Business Media, 2013. 5.4.2.2

[36] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013. 2.5.3

[37] CPN Tools. CPN Tools homepage. `http://www.cpntools.org`. 1.5

[38] CWI and INRIA. The VLTS benchmark suite. `https://cadp.inria.fr/resources/vlts/`, 2019. Last accessed: 2019-05-20. 5.6

[39] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. In *International conference on parallel problem solving from nature*, pages 849–858. Springer, 2000. 5, 5.4.2.2

[40] A. Doudou, B. Garbinato, and R. Guerraoui. Encapsulating failure detection: From crash to byzantine failures. In *International Conference on Reliable Software Technologies*, pages 24–50. Springer, 2002. 2.3.1, 2.3.3

[41] J. J. Durillo and A. J. Nebro. jMetal: A Java framework for multi-objective optimization. *Advances in Engineering Software*, 42(10):760 – 771, 2011. 5, 5.4.2.2

[42] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988. 2.2

[43] M. Ehrgott. *Multicriteria optimization*, volume 491. Springer Science & Business Media, 2005. 5.4.2.2

[44] J. P. Faria and A. C. R. Paiva. A Toolset for Conformance Testing Against UML Sequence Diagrams Based on Event-driven Colored Petri Nets. *Intl. J. on Software Tools for Technology Transfer*, 18(3):285–304, 2016. 4.4

[45] U. Farooq, C. P. Lam, and H. Li. Towards Automated Test Sequence Generation. In *Australian Conf. on Software Engineering (ASWEC 2008)*, pages 441–450, 2008. 4.4

[46] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32(2):374–382, April 1985. 2.2, 2.3.1

[47] P. Fonseca, K. Zhang, X. Wang, and A. Krishnamurthy. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *Proc. of the Twelfth European Conf. on Computer Systems*, pages 328–343. ACM, 2017. 3.6

[48] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, 1988. 1.2.4

[49] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, SOSP '79, pages 150–162, New York, NY, USA, 1979. ACM. 1.1.2

[50] Google Inc. gRPC Remote Procedure Calls. `http://www.grpc.io`. 1.1.3.1

[51] Google Inc. Protocol Buffers. `http://developers.google.com/protocol-buffers`. 1.1.3.1, 3.2

[52] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems (TODS)*, 31(1):133–160, 2006. 1.5, 2.5.2

[53] J. N. Gray. Notes on data base operating systems. In *Operating Systems*, pages 393–481. Springer, 1978. 1.5

[54] W. Grieskamp, N. Kicillof, K. Stobie, and V. Braberman. Model-Based Quality Assurance of Protocol Documentation: Tools and Methodology. *Software Testing, Verification and Reliability*, 21(1):55–71, 2011. 3.6

[55] J. F. Groote and F. van Ham. Interactive visualization of large state spaces. *Intl. Journal on Software Tools for Technology Transfer*, 8(1):77–91, Feb 2006. 5.6

[56] R. Guerraoui, M. Hurfinn, A. Mostéfaoui, R. Oliveira, M. Raynal, and A. Schiper. Consensus in asynchronous distributed systems: A concise guided tour. In *Advances in Distributed Systems*, pages 33–47. Springer, 2000. 2.3.1

[57] R. Guerraoui and M. Raynal. The information structure of indulgent consensus. *IEEE Transactions on Computers*, 53(4):453–466, 2004. 2.3.4

[58] A. Haeberlen, P. Kouznetsov, and P. Druschel. The case for byzantine fault detection. In *HotDep*, 2006. 2.3.3

[59] C. Hawblitzel, J. Howell, M. Kapritsos, J. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. ACM, October 2015. 3.6

[60] K. J. Hayhurst. *A practical tutorial on modified condition/decision coverage*. DIANE Publishing, 2001. 1.2.4

[61] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Trans. Comput. Syst.*, 4:32–53, 02 1986. 1.1.2

[62] X. Huang, J. Wang, J. Qiao, L. Zheng, J. Zhang, and R. K. Wong. Performance and replica consistency simulation for quorum-based nosql system cassandra. In W. van der Aalst and E. Best, editors, *Application and Theory of Petri Nets and Concurrency (PETRI NETS 2017)*, volume 10258 of *LNCS*, pages 78–98. Springer, 2017. 3.6

[63] A. Huima. Implementing Conformiq Qtronic. *TestCom/FATES*, 4581:1–12, 2007. 4.4

[64] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In P. Barham and T. Roscoe, editors, *2010 USENIX Annual Technical Conference*. USENIX Association, 2010. 1, 1.4, 1.8, 3.6

[65] M. Hurfin, A. Mostefaoui, and M. Raynal. Consensus in asynchronous systems where processes can crash and recover. In *Proceedings Seventeenth IEEE Symposium on Reliable Distributed Systems (Cat. No. 98CB36281)*, pages 280–286. IEEE, 1998. 2.3.1

[66] M. Jakobsson and A. Juels. Proofs of work and bread pudding protocols. In *Secure Information Networks*, pages 258–272. Springer, 1999. 2.5.3

[67] K. Jensen and L. Kristensen. Coloured Petri Nets: A Graphical Language for Modelling and Validation of Concurrent Systems. *Comm. ACM*, 58(6):61–70, 2015. 1.5

[68] K. Jensen, L. M. Kristensen, and L. Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, 9(3-4):213–254, 2007. 1.5

[69] P. Jorgensen. *The Craft of Model-based Testing*. CRC Press, 2017. 3.6

[70] P. C. Jorgensen. *Software testing: a craftsman's approach*. Auerbach Publications, 2013. 1.2.1, 1.3, 1.2.3, 1.3.2, 5.3.2

[71] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Solving consensus in a byzantine environment using an unreliable fault detector. In *OPODIS*, volume 97, pages 61–75. Citeseer, 1997. 2.3.1, 2.3.3

[72] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Byzantine fault detectors for solving consensus. *The Computer Journal*, 46(1):16–35, 2003. 2.3.1, 2.3.3

[73] D. E. Knuth. Computer Programming as an Art. *Communications of the ACM*, 17(12):667–673, 1974. I

[74] N. Koochakzadeh and V. Garousi. Tecrevis: a tool for test coverage and test redundancy visualization. In *Intl. Academic and Industrial Conf. on Practice and Research Techniques (TAIC PART)*, volume 6303 of *Lecture Notes in Computer Science*, pages 129–136. Springer, 2010. 5.6

[75] L. M. Kristensen and V. Veiset. Transforming CPN Models into Code for TinyOS: A Case Study of the RPL Protocol. In *Proc. of ICATPN'16*, volume 9698 of *LNCS*, pages 135–154, 2016. 1.5

[76] V. Kuleshov and D. Precup. Algorithms for multi-armed bandit problems. *arXiv preprint arXiv:1402.6028*, 2014. 5.4.1

[77] L. Ladenberger and M. Leuschel. Mastering the visualization of larger state spaces with projection diagrams. In *Formal Methods and Software Engineering - 17th Intl. Conf. on Formal Engineering Methods, ICFEM 2015*, pages 153–169, 2015. 5.6

[78] K. Lakhotia, M. Harman, and P. McMinn. A multi-objective approach to search-based test data generation. In *Proc. of the 9th annual conference on Genetic and evolutionary computation*, pages 1098–1105. ACM, 2007. 5.6

[79] L. Lamport. Proving the correctness of multiprocess programs. *IEEE transactions on software engineering*, (2):125–143, 1977. 2.6

[80] L. Lamport. On interprocess communication. *Distributed computing*, 1(2):86–101, 1986. 2.4

[81] L. Lamport. The Part-time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998. 1.6, 2.5.3

[82] L. Lamport. Paxos Made Simple. *ACM SIGACT News*, 32(4):18–25, December 2001. 1.6, 3.4.2

[83] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982. 2.3.1

[84] A. v. Lamsweerde. Formal specification: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 147–159. ACM, 2000. 1.3.2

[85] M. Larrea, A. Fernández, and S. Arévalo. Eventually consistent failure detectors. In *Proceedings 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing*, pages 91–98. IEEE, 2002. 2.3.4

[86] J. Lawrence, S. Clarke, M. Burnett, and G. Rothermel. How well do professional developers test with code coverage visualizations? An empirical study. In *IEEE Symp. on Visual Languages and Human-Centric Computing*, pages 53–60. IEEE, 2005. 1.2.4, 5.1

[87] T. E. Lea, L. Jehl, and H. Meling. Towards New Abstractions for Implementing Quorum-based Systems. In *Proc. of 37th IEEE Intl. Conf. on Distributed Computing Systems (ICDCS)*, pages 2380–2385, 2017. 1.1.3, 3, 3.4.2, 4.3

[88] W. Linzhang, Y. Jiesong, Y. Xiaofeng, H. Jun, L. Xuandong, and Z. Guoliang. Generating test cases from uml activity diagram based on gray-box method. In *11th Asia-Pacific software engineering conference*, pages 284–291. IEEE, 2004. 1.2.3

[89] J. Liu, X. Ye, and J. Li. Colored Petri Nets Model Based Conformance Test Generation. In *IEEE Symp. on Computers and Communications (ISCC)*, pages 967–970. IEEE, 2011. 4.4

[90] S. Lu, P. Zhou, W. Liu, Y. Zhou, and J. Torrellas. Pathexpander: Architectural support for increasing the path coverage of dynamic bug detection. In *Proc. of the 39th Annual IEEE/ACM Intl. Symp. on Microarchitecture*, pages 38–52. IEEE Computer Society, 2006. 1.2.4

[91] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler. GRT: program-analysis-guided random testing. In *Proc. 30th Intl. Conf. on Automated Software Engineering (ASE 2015)*, pages 212–223, Lincoln, USA, 2015. IEEE. 5.6

[92] F. Macias, T. Scheffel, M. Schmitz, and R. Wang. Integration of runtime verification into metamodeling for simulation and code generation (position paper). In *Runtime Verification*, volume 10012 of *Lecture Notes in Computer Science*, pages 454–461. Springer International Publishing, 2016. 1.9

[93] F. Macıas, T. Scheffel, M. Schmitz, R. Wang, M. Leucker, A. Rutle, and V. Stolz. Integration of Runtime Verification into Metamodeling. In *Proceedings of the 28th Nordic Workshop on Programming Theory (NWPT), Aalborg University, Denmark*, 2016. 1.9

[94] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, Oct 1998. 1.1.2

[95] C. Martín and M. Larrea. Eventual leader election in the crash-recovery failure model. In *2008 14th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 208–215. IEEE, 2008. 2.3.4

[96] MBT/CPN repository. `https://github.com/selabhvl/mbtcpn`, Aug 2018. 3.3, 3.4

[97] H. Meling and L. Jehl. Tutorial Summary: Paxos Explained from Scratch. In *17th International Conference on Principles of Distributed Systems (OPODIS)*, pages 1–10, 2013. 1, 1.1.3, 1.6, 2.5.3, 3.4.2

[98] D. Mondal, H. Hemmati, and S. Durocher. Exploring test suite diversification and code coverage in multi-objective test case selection. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10. IEEE, 2015. 5.6

[99] A. Mostefaoui and M. Raynal. Leader-based consensus. *Parallel Processing Letters*, 11(01):95–107, 2001. 2.3.4

[100] G. J. Myers, T. Badgett, T. M. Thomas, and C. Sandler. *The art of software testing*, volume 2. Wiley Online Library, 2004. 5

[101] A. J. Nebro, J. J. Durillo, and M. Vergne. Redesigning the JMetal multi-objective optimization framework. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO Companion '15, pages 1093–1100, New York, NY, USA, 2015. Association for Computing Machinery. 5, 5.4.2.2

[102] C. *et al.*. Newcombe. How amazon web services uses formal methods. *Commun. ACM*, 58(4):66–73, Mar. 2015. 1, 2.5.3

[103] R. Oliveira, R. Guerraoui, and A. Schiper. Consensus in the crash-recover model. *EPFL, Dept. d'Informatique, Tech. rep*, pages 97–239, 1997. 2.3.1

[104] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319, 2014. 1, 2.5.3

[105] N. Oster and F. Saglietti. Automatic test data generation by multi-objective optimisation. In *International Conference on Computer Safety, Reliability, and Security*, pages 426–438. Springer, 2006. 5.6

[106] O. Padon, G. Losa, M. Sagiv, and S. Shoham. Paxos Made EPR: Decidable Reasoning about Distributed Protocols. *Proceedings of the ACM on Programming Languages*, 1:108:1–108:31, October 2017. 3.6

[107] H. Ponce de León, S. Haar, and D. Longuet. Model-based Testing for Concurrent Systems: Unfolding-based Test Selection. *International Journal on Software Tools for Technology Transfer*, 18(3):305–318, 2016. 3.6

[108] Programming Methods Laboratory of École Polytechnique Fédérale de Lausanne. The Scala Programming Language. `https://www.scala-lang.org`. 1.4

[109] A. Rajan, M. W. Whalen, and M. P. Heimdahl. The effect of program and model structure on mc/dc test adequacy coverage. In *Proceedings of the 30th international conference on Software engineering*, pages 161–170. ACM, 2008. 1.2.4

[110] W. Reisig. *Petri nets: an introduction*, volume 4. Springer Science & Business Media, 2012. 1.5

[111] W. Reisig. *Elements of distributed algorithms: modeling and analysis with Petri nets*. Springer Science & Business Media, 2013. 1.5

[112] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri. Combining multiple coverage criteria in search-based unit test generation. In M. Barros and Y. Labiche, editors, *Search-Based Software Engineering*, pages 93–108. Springer, 2015. 5.6

[113] A. Rountev, S. Kagan, and J. Sawin. Coverage criteria for testing of object interactions in sequence diagrams. In *Fundamental Approaches to Software Engineering, 8th Intl. Conf., FASE 2005*, pages 289–304, 2005. 5.6

[114] A. Saifan and J. Dingel. Model-based Testing of Distributed Systems. Technical Report 548, School of Computing, Queen's University, Canada, 2008. 3.6

[115] A. Salahirad, H. Almulla, and G. Gay. Choosing the fitness function for the job: Automated generation of test suites that detect real faults. *Softw. Test., Verif. Reliab.*, 29(4-5), 2019. 5.6

[116] R. D. Schlichting and F. B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems (TOCS)*, 1(3):222–238, 1983. 2.3.1

[117] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22:299–319, December 1990. 1.1.1

[118] C. Shao, J. L. Welch, E. Pierce, and H. Lee. Multiwriter consistency conditions for shared memory registers. *SIAM Journal on Computing*, 40(1):28–62, 2011. 2.4

[119] D. Skeen. Nonblocking commit protocols. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 133–142. ACM, 1981. 2.5.2

[120] Standard ML of New JerseyCPN Tools. `http://www.smlnj.org`. 1.5

[121] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. Cambridge, MA: MIT Press, 2011. 5.4.1

[122] G. Tretmans and H. Brinksma. TorX: Automated Model-Based Testing. In A. Hartman and K. Dussa-Ziegler, editors, *1st Europ. Conf. on Model-Driven Software Engineering*, pages 31–43, 12 2003. 4.4

[123] D. Tziatzios. Model-based testing for SQL databases. Master's thesis, KTH, School of Electrical Engineering and Computer Science (EECS), 2019. 5.5

[124] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Elsevier, 2010. 1.3.1, 1.4, 6.1

[125] M. Utting, A. Pretschner, and B. Legeard. A Taxonomy of Model-based Testing Approaches. *Software Testing, Verification and Reliability*, 22:297–312, 2012. 1, 1.3.2, 1.3.2, 1.5, 1.3.2, 1.4, 5.4.2.1

[126] M. Vukolić. Quorum Systems: With Applications to Storage and Consensus. *Synthesis Lectures on Distributed Computing Theory*, 3(1):1–146, 2012. 1.1.2, 1.1.3

[127] M. Vukolić et al. The origin of quorum systems. *Bulletin of EATCS*, 2(101), 2013. 1.1.3

[128] R. Wang, C. Artho, L. M. Kristensen, and V. Stolz. Visualization and Abstractions for Execution Paths in Model-Based Software Testing. In *Integrated Formal Methods*, volume 11918 of *Lecture Notes in Computer Science*, pages 474–492. Springer International Publishing, 2019. 1.8, 1.9, 5, 5.3, 5.3.2, 5.3.2.2, 5.4.2.1, 5.5, 5.5, 5.6, 6.1, 6.2.1

[129] R. Wang, C. Artho, L. M. Kristensen, and V. Stolz. Multi-objective Search for Model-based Testing. In *The 20th IEEE International Conference on Software Quality, Reliability, and Security*, Vilnius, Lithuania, 2020. IEEE. 1.8, 1.9, 5, 5.4.1, 5.4.1, 5.4.1, 5.4.2.2, 5.5, 5.5, 5.5, 5.6, 6.1

[130] R. Wang, L. M. Kristensen, H. Meling, and V. Stolz. Application of model-based testing on a quorum-based distributed storage. In *CEUR Workshop Proceedings, Petri Nets and Software Engineering (PNSE'17)*, volume 1846, pages 177–196, 2017. 1.9

[131] R. Wang, L. M. Kristensen, H. Meling, and V. Stolz. Model-based Testing of the Gorums Framework for Fault-tolerant Distributed Systems. In *Proceedings of the 29th Nordic Workshop on Programming Theory (NWPT), Turku Centre for Computer Science, Finland*, 2017. 1.9

[132] R. Wang, L. M. Kristensen, H. Meling, and V. Stolz. Model-Based Testing of the Gorums Framework for Fault-Tolerant Distributed Systems. In *Transactions on Petri Nets and Other Models of Concurrency XIII*, volume 11090 of *Lecture Notes in Computer Science*, pages 158–180. Springer International Publishing, 2018. 1.8, 1.9, 3, 3.1, 3.3, 3.4, 3.4.1, 3.5, 4, 4.3, 4.3, 4.4, 6.1

[133] R. Wang, L. M. Kristensen, H. Meling, and V. Stolz. Automated test case generation for the Paxos single-decree protocol using a Coloured Petri Net model. In *Journal of Logical and Algebraic Methods in Programming*, volume 104, pages 254–273. Elsevier Ltd, 2019. 1.8, 1.9, 3, 3.2, 3.3, 3.4, 3.4.2, 3.5, 3.5, 4, 4.3, 4.3, 4.4, 6.1

[134] R. Wang, L. M. Kristensen, and V. Stolz. MBT/CPN: A Tool for Model-Based Software Testing of Distributed Systems Protocols Using Coloured Petri Nets. In *Verification and Evaluation of Computer and Communication Systems*, volume 11181 of *Lecture Notes in Computer Science*, pages 97–113. Springer International Publishing, 2018. 1.8, 1.9, 4, 4.3, 6.1

[135] H. Watanabe and T. Kudoh. Test Suite Generation Methods for Concurrent Systems Based on Coloured Petri Nets. In *Software Engineering Conference*, pages 242–251. IEEE, 1995. 3.6

[136] D. Wu, E. Schnieder, and J. Krause. Model-based Test Generation Techniques Verifying the On-board Module of a Satellite-based Train Control System Model. In *2013 IEEE Intl. Conf. on Intelligent Rail Transportation Proceedings*, pages 274–279, Aug 2013. 4.4

[137] D. Xu. A Tool for Automated Test Code Generation from High-level Petri Nets. In *Proc. of ICATPN'2011*, volume 6709 of *LNCS*, pages 308–317. Springer, 2011. 4.4

[138] S. Yoo and M. Harman. Pareto efficient multi-objective test case selection. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 140–150, 2007. 5.6

[139] J. Zander, I. Schieferdecker, and P. J. Mosterman. *Model-based testing for embedded systems*. CRC press, 2011. 1.2.3

[140] W. Zheng, C. Liang, R. Wang, and W. Kong. Automated Test Approach Based on All Paths Covered Optimal Algorithm and Sequence Priority Selected Algorithm. *IEEE Transactions on Intelligent Transportation Systems*, 15(6):2551–2560, 2014. 3.6

[141] H. Zhu, P. A. Hall, and J. H. May. Software unit test coverage and adequacy. *Acm computing surveys (csur)*, 29(4):366–427, 1997. 1.2.4

Part II

ARTICLES

PAPER A

MODEL-BASED TESTING OF THE GORUMS FRAMEWORK FOR FAULT-TOLERANT DISTRIBUTED SYSTEMS

R. Wang, L. M. Kristensen, H. Meling, and V. Stolz.

In Transactions on Petri Nets and Other Models of Concurrency XIII, volume 11090 of *Lecture Notes in Computer Science*, pages 158–180, Springer International Publishing, 2018.

Model-Based Testing of the Gorums Framework for Fault-Tolerant Distributed Systems

Rui Wang[1(✉)], Lars Michael Kristensen[1], Hein Meling[2], and Volker Stolz[1]

[1] Department of Computing, Mathematics, and Physics,
Western Norway University of Applied Sciences, Bergen, Norway
{rwa,lmkr,vsto}@hvl.no
[2] Department of Electrical Engineering and Computer Science,
University of Stavanger, Stavanger, Norway
hein.meling@uis.no

Abstract. Data replication is a central mechanism for the engineering of fault-tolerant distributed systems, and is used in the realization of most cloud computing services. This paper explores the use of Coloured Petri Nets (CPNs) for model-based testing of quorum-based distributed systems. We have developed an approach to model-based testing of fault-tolerant services implemented using the Go language and the Gorums framework. We show how a CPN model can be used to obtain both unit test cases for the quorum logic functions, and system level test cases consisting of quorum calls. The CPN model is also used to obtain the test oracles against which the result of running a test case can be compared. We demonstrate the application of our approach by considering an implementation of a distributed storage service on which we obtain 100% code coverage for the quorum functions, 96.7% statement coverage on the quorum calls, and 52.3% coverage on the Gorums framework. We demonstrate similar encouraging results also on a more complex Gorums-based implementation of the Paxos consensus protocol.

1 Introduction

Distributed systems serve millions of users in many important applications and domains. However, such complex systems are known to be difficult to implement correctly because they must cope with challenges such as concurrency and failures [12]. Thus, when designing and implementing distributed systems, it is important to ensure correctness and fault-tolerance. Distributed systems can rely on a quorum system to achieve fault-tolerance, yet it remains challenging to implement fault-tolerance correctly. Therefore, the use of testing techniques is essential to detect bugs and to improve the correctness of such systems.

One promising testing approach is *model-based testing* (MBT) [23]. MBT is a paradigm based on using models of a system under test (SUT) and its environment to generate test cases for the system. The goal of MBT is validation and

M. Koutny et al. (Eds.): ToPNoC XIII, LNCS 11090, pp. 158–180, 2018.
https://doi.org/10.1007/978-3-662-58381-4_8

error-detection by finding observable differences between the behavior of the implementation and the intended behavior of the SUT. A test case consists of test input and expected output and can be executed on the SUT. Typically, MBT involves: (a) build models of the SUT from informal requirements; (b) define test selection criteria for guiding the generation of test cases and the corresponding test oracle representing the ground-truth; (c) generate and run test cases; (d) compare the output from test case execution with the expected result from the test oracle. The component that performs (c) and (d) is known as a *test adaptor* and uses a *test oracle* to determine whether a test has passed or failed.

In this paper, we investigate the use of Coloured Petri Nets (CPNs) [11] for model-based testing applied to quorum-based distributed systems [24]. Quorum systems are fundamental to building fault-tolerant distributed systems, and recently the Gorums framework [17] has been developed to ease the implementation of quorum-based distributed systems. The Gorums framework constitutes a distributed middleware that hides the complexity in implementing the communication, synchronization, message processing, and error handling between the protocol entities. The widespread use of the Gorums framework will depend on the correctness of its implementation in Go. This motivates our goal of systematically testing the Gorums middleware implementation and provides an MBT approach that can be used to also systematically test applications that rely on the Gorums framework implementation.

The contribution of this paper is to propose an MBT approach using CPNs for quorum-based distributed applications implemented by the Gorums framework. To illustrate the application of our approach, we show in detail how it can be used on a Gorums-based implementation of a single-writer, multi-reader distributed storage. The distributed storage system is implemented with a read and a write *quorum call*, which clients can use to access the distributed storage. The distributed storage may return multiple replies to a quorum call. To simplify client access to the storage, Gorums uses a user-defined *quorum function* to coalesce the replies into a single reply that can then be returned to the client. For this particular storage system, we use a majority quorum. By developing a CPN model of such a distributed storage, we are able to generate test cases consisting of read and write quorum calls that test the Gorums framework implementation. For evaluation, we report on results obtained on the distributed storage system, and present results obtained on a more complex example in the form of the Paxos consensus protocol [16].

CPNs has been widely used for modeling and verifying models of distributed systems spanning domains such as workflow systems, communication protocols, and distributed algorithms [14]. Recently, work has also been done on automated code generation allowing an implementation of the modeled systems to be obtained [15]. Comprehensive testing of an implementation is, however, an equally important task in the engineering of distributed systems, independently of how the implementation has been obtained. This also applies in the case of automated code generation, as it is seldom the case that the correctness of the model-to-text transformations and their implementation can be formally proved. We have chosen CPNs as the foundation of our MBT approach as it has a strong track record for modeling distributed systems, and enables compact modeling

of data and data manipulation which is required for message modeling, quorum functions modeling, and concrete test case generation. Furthermore, CPNs has the ability to create parametric models, perform model validation prior to test case generation, and it has mature tool support for both simulation and state space exploration, which is important in order to implement our approach and conduct practical experiments.

The rest of this paper is organized as follows. Section 2 introduces quorum-based distributed systems and the Gorums framework, and Sect. 3 describes the Gorums-based distributed storage which constitutes our system under test. Section 4 presents the constructed CPN model for test case generation, and Sect. 5 shows how state-spaces can be used to obtain test cases and test oracles. In Sect. 6 we present the Go implementation of our test adapter and how it is connected to the Gorums implementation of the distributed storage in order to execute the test cases. In Sect. 7 we report on experimental results. Section 8 presents related work, and in Sect. 9 we sum up conclusions and present directions for future work. The reader is assumed to be familiar with the basic concepts of high-level Petri Nets. This paper is an extended and revised version of an earlier workshop paper [25].

2 Quorum-Based Distributed Systems and Gorums

Distributed algorithms are commonly used to implement replicated services, and they rely on a quorum system [24] to achieve fault tolerance. That is, to access the replicated state, a process only needs to contact a quorum, e.g. a majority of the processes. In this way, a system can provide service despite the failure of individual processes. However, communicating with and handling replies from sets of processes often complicate the protocol implementations. The Gorums [17] framework has been developed to alleviate the development effort for building advanced distributed algorithms, such as Paxos [16] and distributed storage [2].

The Gorums framework reduces the complexity of implementing quorum-based distributed systems by providing two core abstractions: (a) a flexible and simple quorum call abstraction, which is used to communicate with a set of processes and to collect their responses, and (b) a quorum function abstraction which is used to process responses. These abstractions help to simplify the main control flow of protocol implementations. Figure 1 illustrates the interplay between the main abstractions provided by Gorums. Gorums consists of a run-time library and code generator that extends the gRPC [8] remote procedure

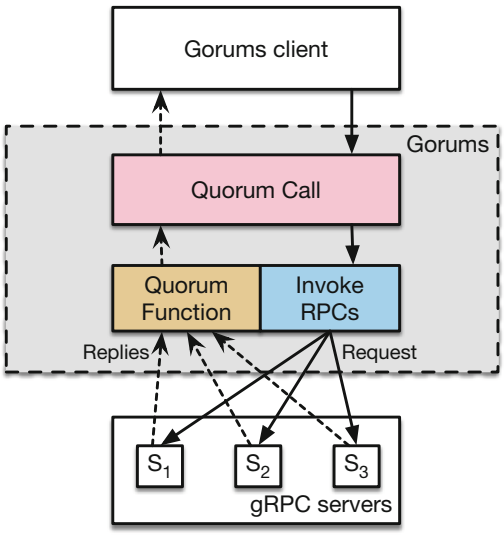


Fig. 1. Gorums architecture.

call library. Gorums allows clients to invoke a quorum call, i.e. a set of RPCs, on a group of servers, and to collect their replies. The replies are processed by a quorum function to determine if a quorum has been obtained. The quorum function is invoked every time a new reply is received at the client, to evaluate whether or not the received set of replies constitutes a quorum. With Gorums, developers can specify several RPC service methods using `protobuf` [9], and from this specification, Gorums's code generator will produce code to facilitate quorum calls and collection of replies. However, each RPC/quorum call method must provide a user-defined quorum function that Gorums will invoke to determine if a quorum has been obtained for that specific quorum call. In addition, the quorum function also provides a single reply value, based on a coalescing of the received reply values from the different server replicas. This coalesced reply value is then returned to the client as the result of its quorum call. That is, the invoking client does not see the individual replies.

The contribution of this paper is to provide an MBT approach for generating test cases to validate the correctness of the Gorums framework implementation itself. This comes in addition to test cases for quorum function and quorum call implementations for a specific use of the framework such as for implementing a distributed storage. The quorum functions for a specific protocol implementation must follow a well-defined interface generated by Gorums. These only require a set of reply values as input and a return of a single reply value together with a boolean quorum decision. Hence, quorum functions can easily be tested using unit tests. However, some quorum functions involve complex logic, and their input and output domains may be large, and so generating test cases from a model provides significant benefit to verify correctness. A quorum call is implemented by a set of RPCs, invoked at different servers, and so different interleavings must be considered due to invocations by different clients. Hence, using MBT we can produce sequences of interleavings aimed at finding bugs in the server-side implementations of the RPC methods and also in the Gorums runtime system.

3 System Under Test: Gorums and Distributed Storage

We have implemented a distributed storage system, with a single writer and multiple readers. The storage system is replicated for fault-tolerance, and is implemented using Gorums. To test this storage implementation, we have designed a corresponding CPN model that we use to generate test cases (see Sect. 4). In this section, we describe the different components of the distributed storage and how it has been implemented using Gorums.

As with any RPC library, Gorums requires that the server implements the methods specified in the service interface. For our distributed storage, we have implemented two server-side methods: Read() and Write(). These can be invoked as quorum calls from storage clients, to read/write the state of the storage. In our current implementation, we allow only a single write quorum call to be invoked, but any number of read quorum calls can be invoked by the client to read the state of the storage.

A client reading from the storage may observe different replies returned by the different server replicas. The reason for this is that the read may be interleaved with one or more writes generated by the client. To allow a reader to pick the correct reply value to return from a quorum call, each server maintains a timestamp that is incremented for each new Write(). That is, the reader will always return the value associated with the reply with the highest timestamp. Thus, to implement the reader using Gorums, we can simply implement a user-defined ReadQF quorum function for the Read() quorum call as shown in Algorithm 1. As this code illustrates, a set of replies from the different servers are coalesced into a single reply that can then be returned from the quorum call. The reply of the quorum function is determined by the reply from the server(s) having the highest timestamp.

The user-defined quorum functions are implemented as methods on an object of type QuorumSpec, named *qs* in Algorithm 1. This object holds information about the quorum size, such as ReadQSize, and other parameters used by the quorum functions. This *qs* object must satisfy an interface generated by Gorums's code generator. In Algorithm 1, ReadQSize is used to determine if sufficient replies have been received to return the server reply with the highest timestamp.

Algorithm 1. Read quorum function

```
1: func (qs QuorumSpec) ReadQF(replies []ReadReply)
2:     if len(replies) < qs.ReadQSize then                    ▷ read quorum size
3:         return nil, false                  ▷ no quorum yet, await more replies
4:     highest := ⊥                           ▷ reply with highest timestamp seen
5:     for r := range replies do
6:         if r.Timestamp ≥ highest.Timestamp then
7:             highest := r
8:     return highest, true                                   ▷ found quorum
```

4 CPN Testing Model for the Distributed Storage

In this section, we describe the CPN model of our test framework developed in order to generate test cases for the Gorums framework and the distributed storage implementation presented in Sect. 3. We model the entire system, parametrized by the number of clients and servers. Some key features of the model are the use of colored tokens for distinguishing multiple incoming and outgoing messages, and the quorum specification based on the numbers of replies received so far.

Figure 2 shows the top-most module of the CPN model. The substitution transition Clients represents the clients (users) of the distributed storage system while Servers represent the servers. The places ClientToServer and ServerToClient are used for modeling the message channels for communication between the clients and the servers. The CPN model has been constructed in a folded manner

so that the number of servers is a parameter that can be configured without making changes to the net-structure. Below we provide more details on selected modules of the CPN model. The complete CPN model including all color sets, variable declarations, and function definitions is available from [5].

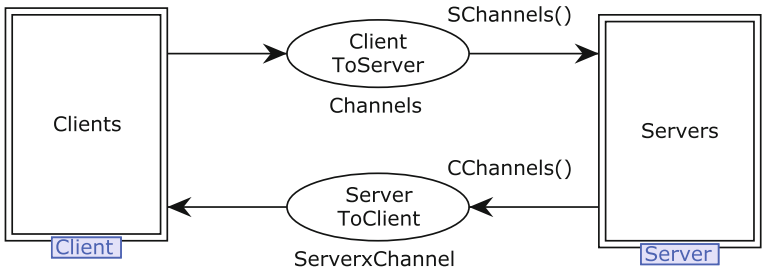


Fig. 2. Top-level module of the CPN model.

Figure 3 shows the client submodule of the Clients substitution transition in Fig. 2. The substitution transition QuorumCalls is used to model the behavior of applications running on the clients, which makes the read and write quorum calls. In particular, the submodules of QuorumCalls serve as test driver modules used to generate system tests for the distributed storage and the Gorums framework. The content of QuorumCalls depends on the specific test scenarios to be investigated for the system under test, and we give a concrete example of a test driver module in Sect. 6. The substitution transitions Read and Write represent the quorum calls provided by the distributed storage. The invocation of quorum calls is done by placing tokens on the Read and Write places. The port places ServerToClient and ClientToServer are linked to the identically named socket places in Fig. 2.

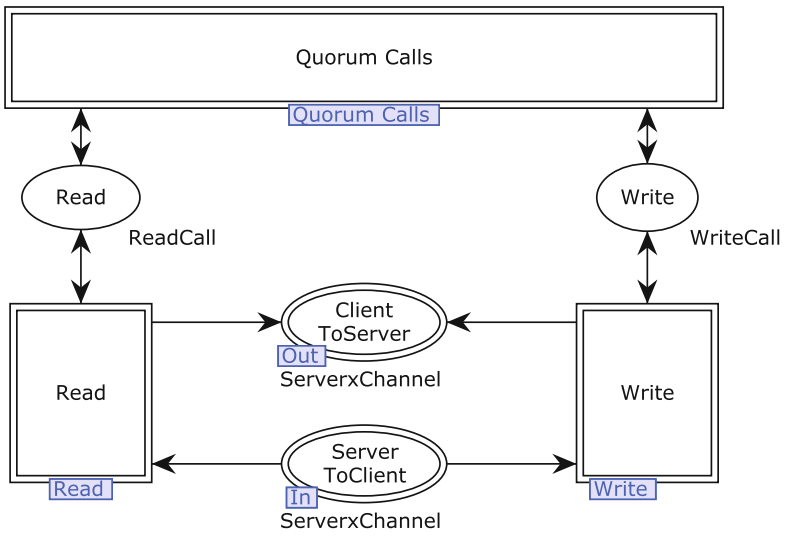


Fig. 3. The Clients module.

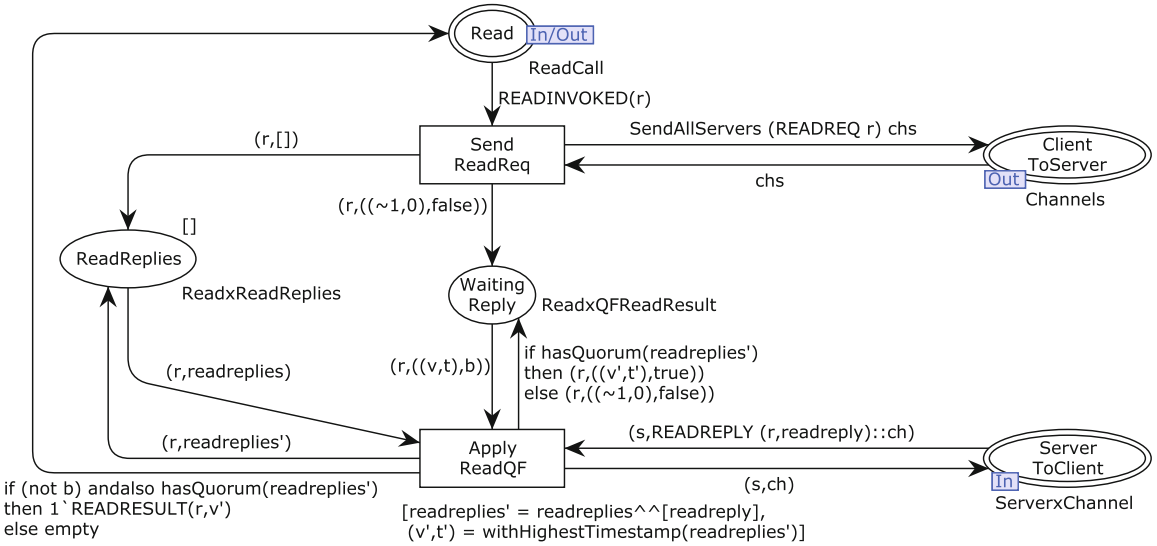


Fig. 4. The Read module.

Figure 4 shows the submodule of the Read substitution transition which provides an abstract implementation of the Read() quorum call. The main purpose of the Read module is to generate test cases for the ReadQF quorum function. A read quorum call is triggered by the presence of a token with the color `READINVOKED(r)`, where `r` identifies the call and is used to match replies from servers to the call. The execution of a read quorum call starts by sending a read request to each of the servers. This is modeled by the transition SendReadReq and the expression on the arc to place ClientToServer, which will add tokens representing read requests being sent to the servers. In addition, a list-token is put on place ReadReplies, which is used to collect the replies received from the servers. The call then enters a WaitingReply state and waits for replies coming back from the servers. When a read's reply comes back, represented by a token on place ServerToClient, then transition ApplyReadQF will be enabled. This transition takes the current list of `readreplies` and appends the received `readreply` to form `readreplies'`. The quorum function is then invoked, as represented by the arc expressions to WaitingReply and Read. If enough replies have been received, then a read result is returned to the Read place containing the value with the highest timestamp. As we will see later, we use occurrences of the ApplyReadQF transition for generating test cases for the ReadQF quorum function. In addition, we record the result computed by the CPN model as the test oracle and compare it to the result of our SUT's implementation of the ReadQF quorum function. The submodule for the Write() quorum call is similar. It has a transition ApplyWriteQF, which we use as a basis for generating test cases and obtain a test oracle for the WriteQF quorum function.

Figure 5 shows the server submodule of the Servers substitution transition in Fig. 2. The replicated state of each server is modeled by the place State. The two substitution transitions are used for modeling the handling of write requests and read requests on the server side.

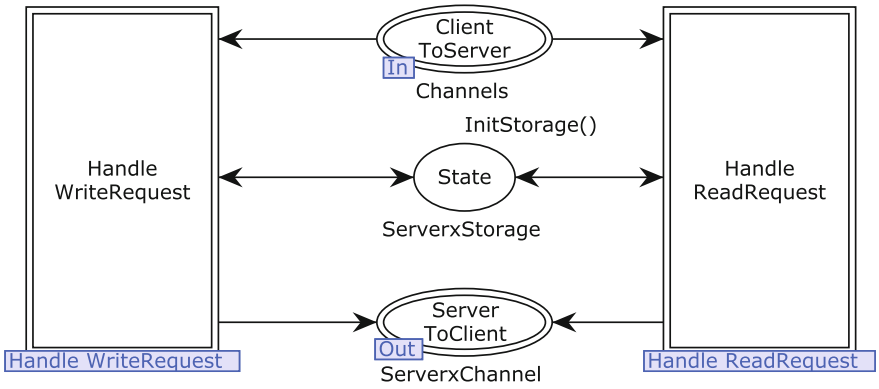


Fig. 5. The Server module.

Figure 6 shows the submodule of the substitution HandleWriteRequest modeling the processing of a write request from a client. The incoming write request will be presented as a value in the list-token on place ClientToServer and contains a value `v'` to be written in the distributed storage together with a timestamp `t'`. The server compares the timestamp of the incoming write request with the timestamp `t` for the currently stored value `v`. If the timestamp of the incoming write request is larger, then the new value is stored on the server, and a write acknowledgment is sent back in a write reply to the client. Otherwise, the stored value remains unchanged and a negative write acknowledgment is sent to the client in the write reply. Handling of an incoming request requires that the server is `running` (as opposed to `failed`) as modeled by the double arc connecting ServerStatus and HandleWriteRequest. The handling of read requests is modeled in a similar manner, except that no comparison is needed, and the server simply returns the currently stored value together with its timestamps.

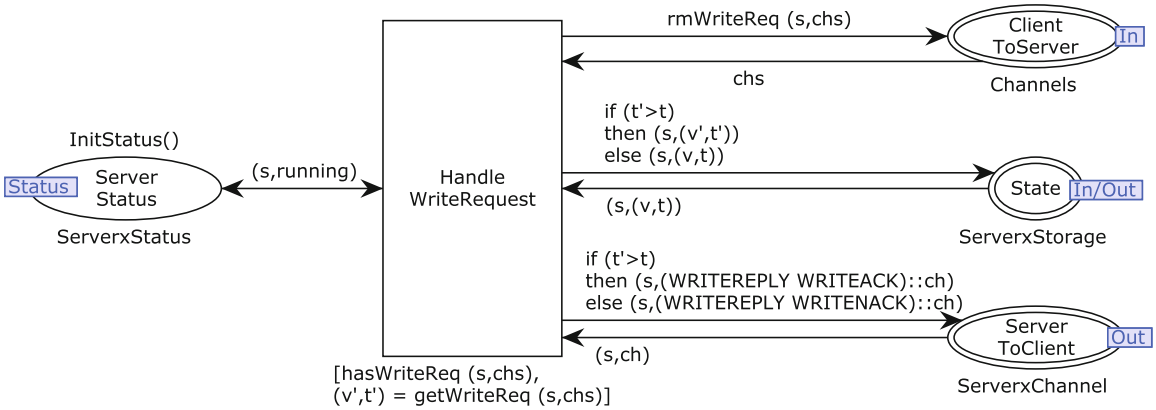


Fig. 6. The HandleWriteRequest module.

5 Test Case Generation

The generation of test cases for Gorums and the distributed storage system is based on the analysis of executions of the CPN model. Test cases can be generated for both the quorum functions and the quorum calls.

The test cases generated for the quorum functions are unit tests, whereas the test cases generated for quorum calls are system tests consisting of concurrent and interleaved invocations of read and write quorum calls. The latter tests both the implementation of the quorum calls and the Gorums framework implementation. In addition to the test cases, we also generate a *test oracle* for each test case to determine whether the test has passed.

5.1 Unit Tests for Quorum Functions

Test cases for the ReadQF quorum function can be obtained by considering occurrences of the ApplyReadQF transition (Fig. 4). When this transition occurs, the variable `readreplies'` is bound to the list of all replies that have been received from the servers so far, and which the quorum function is invoked on. In addition, we can use the implementation of the quorum function in the CPN model as the test oracle. This means that the expected result of invoking the quorum function can be obtained by considering the value of the token put back on place WaitingReply. The value of this token contains the result of invoking the quorum function in its second component. Generally, occurrences of ApplyReadQF can be detected using either state spaces or simulations:

State-space based detection. We explore the full state space of the CPN model searching for arcs corresponding to the ApplyReadQF transition. Whenever an occurrence is encountered we emit a test case together with the expected result. In this case, we obtain test cases for all the possible ways in which the quorum function can be invoked in the CPN model.

Simulation-based detection. We run a simulation of the CPN model and use the monitoring facilities of the CPN Tools [4] simulator to detect occurrences of the ApplyReadQF transition and emit the corresponding test cases. The advantage of this approach over the state-space based approach is scalability, while the disadvantage is potentially reduced test coverage.

Test cases are generated based on detecting transition occurrences. This is done in a uniform way for both detection approaches. Specifically, we rely on a *detection function*, which must evaluate to true whenever a specific transition occurrence is detected. When this happens, a *generator function* is invoked to generate the actual test case. The state space for the CPN testing model of the distributed storage service is relatively small and we can obtain all test cases based on state space-based detection. The Paxos consensus protocol considered in Sect. 7 is more complex, and hence we rely on simulation-based detection for its test case generation.

Listing 1 shows an example of how our test cases are represented using XML. The test case for the ReadQF quorum function corresponds to two replies (one with value 0 and timestamp 0, and one with value 42 and timestamp 1). With three servers, this constitutes a quorum, and the value returned from the quorum function is therefore expected to be 42 with the timestamp of 1.

```
<Test TestName="ReadQFTest">
  <TestCase CaseID="1">
    <TestValues>
      <Content>
        <Value>0</Value>
        <Timestamp>0</Timestamp>
      </Content>
      <Content>
        <Value>42</Value>
        <Timestamp>1</Timestamp>
      </Content>
    </TestValues>
    <ExpectResults>
      <Value>42</Value>
      <Timestamp>1</Timestamp>
    </ExpectResults>
    <ExpectQuorum>true</ExpectQuorum>
  </TestCase>
</Test>
```

Listing 1. Example of generated test cases for read quorum function.

5.2 System Tests of Quorum Calls

The generation of test cases and expected results is based on the submodule of the QuorumCalls substitution transition (see Fig. 3). This module acts as a test driver for the system by specifying scenarios for read and write quorum calls to the underlying quorum system. By varying this module, it is possible to generate different scenarios of read and write quorum calls.

Figure 7 shows an example of a test driver in which the client executes one read and one write quorum call as modeled by the transition InvokeRDWR. Upon completion of these two calls, there are server failures and a new read and a write call is invoked (modeled by the transition InvokeRDWRFailures). The server failures are modeled by changing the color of the server-tokens on place ServerStatus which is used with the place ServerStatus on the HandleWriteRequest (see Fig. 6). Each quorum call has a unique identifier (1, 2, 3, and 4) for identifying the call. Each write call also has a value (in this case 42 and 7) to be written to the distributed storage.

To make test case generation independent to the particular test driver module, we exploit that the read and write quorum calls, made during an execution of the CPN model, can be observed as tokens on the Read and Write socket places (see Fig. 3). When there is a `READINVOKED(i)` token on place READ for some integer `i`, it means that a read quorum call identified by `i` has been invoked. When the read quorum call has terminated, there will be a token with the color `READRESULT(i,v)` present on the place Read, where `v` is the value read by the call. The invocation and termination of write quorum calls can be detected in a similar manner by considering the tokens with the colors `WRITEINVOKED(i,v)`

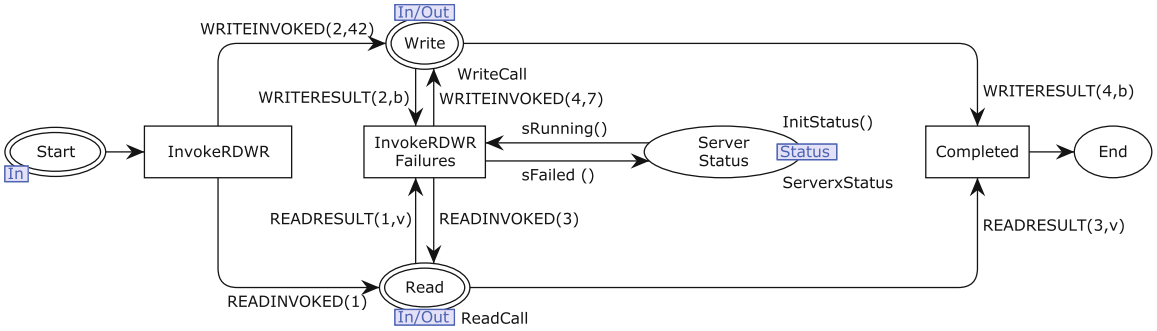


Fig. 7. The QuorumCalls module.

and `WRITERESULT(i,b)` on the place Write (Fig. 3), where the boolean value `b` denotes whether the value `v` was written or not.

Based on this, we can generate test cases in XML format specifying both the concurrent and sequential execution of read and write calls. Listing 2 (discussed further below) shows an example where first a read and a write are initiated and upon completion of these two calls, a new read call is initiated.

```
<Test TestName="SystemTest">
  <TestCase CaseID="WRprRDsqRD">
    <Routine RoutineID="A" OperationName="Write">
      <OperationValues>
        <Value>7</Value>
      </OperationValues>
      <Routine RoutineID="B" OperationName="Read">
        <OperationValues>
           <Value>7</Value>
           <Value></Value>
        </OperationValues>
      </Routine>
   </Routine>
   <Routine RoutineID="A" OperationName="Read">
     <OperationValues>
       <Value>7</Value>
     </OperationValues>
   </Routine>
 </TestCase>
</Test>
```

Listing 2. Example of a generated test cases for the concurrent and sequential execution of read and write calls.

We handle concurrent executions by nesting the read and write `Routine` tag as illustrated in Listing 2, while non-nested `Routine` tags are considered sequential. For write calls, we use the value tag to specify the value to be written, and for read calls we use the value tag to describe permissible values for the test case (see next section) returned by read calls. The absence of a value between value tags indicates that the result could be null—corresponding to the case where no value have yet been written into the storage.

It should be noted if the CPN model specifies that a read and write call may execute concurrently (independently), but happened to be executed in sequence in a concrete execution of the CPN model (e.g., first the read executes and completes and then the write executes an completes), then that will be specified as a sequential test case in the XML format. This is not a problem as the CPN model captures all the possible executions and hence there will be another execution of the CPN model in which the read and the write are running concurrently.

5.3 Test Oracle for System Tests

Checking that the result of an execution with read and write quorum calls is as expected is more complex than for quorum functions. This is because the result of concurrently executing read and write calls depends on the order in which messages are sent and received. Figure 8 shows an example test case in which there are two routines (threads of execution) that concurrently execute read and write quorum calls. When $Write_1$ and $Read_a$ are initialized and executed concurrently, the returned result of $Read_a$ could be the old value in the servers before $Write_1$ writes a new value to servers, or the returned result of $Read_a$ could be the value already written by $Write_1$. The same situation applies to $Write_2$ and $Read_c$. Since they are executed concurrently, the returned value of $Read_c$ could be the value written by $Write_1$ or $Write_2$.

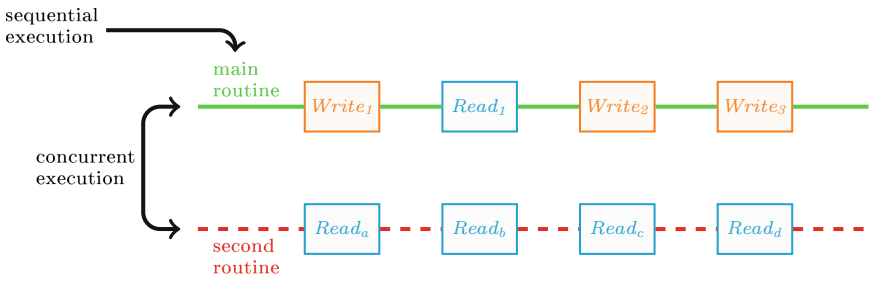


Fig. 8. An example of concurrent and sequential execution of quorum calls.

This means that if we execute (simulate) the CPN model with a test case containing concurrent read and write quorum calls, then the result returned upon completion of the calls may be different if we execute the same test case against the Go implementation. The reason is that we cannot control in what order the messages are sent and delivered by the underlying gRPC library, i.e., due to non-determinism in the execution. When we apply a state-space based approach for extracting the test cases, e.g., for the quorum function, then we can compute all the possible legal outcomes of a quorum call since the state space captures all interleaved executions. In contrast, we cannot obtain all legal values when extracting test cases from a single execution of the CPN model.

The first step towards constructing a general test oracle is to characterize the permissible values of a read quorum call. These are:

1. the initial value of the storage in case no writes were invoked before the read was invoked, or;

2. the value of the most recent write invoked but not terminated prior to the read call (if any) or;
3. the value of the most recent write that has terminated prior to invocation of the read or;
4. the value of a write that was invoked between the invocation and completion of the read.

The above can be formally captured in the stateful automaton shown in Fig. 9 (left), which can be used to monitor the global correctness of the distributed storage. The four events are shorthands for the abstract tokens per client-request observed in the model, e.g., `READINVOKED(i)` is abbreviated RI_i.

The set S is used to collect the set of permissible values for a read call. On a read call RI_i, any pending write $WI(c)$ observed since the last write-return $WI(c)$ is a potential read-result. We abuse notation from alternating automata with parametrized propositions [22] to capture that on a read invocation, we remain in the initial state and collect further input for a new instance of the monitor with the same current state (indicated by the dashed line) for subsequent read-invocations. We explain Fig. 9 (right) in the next section.

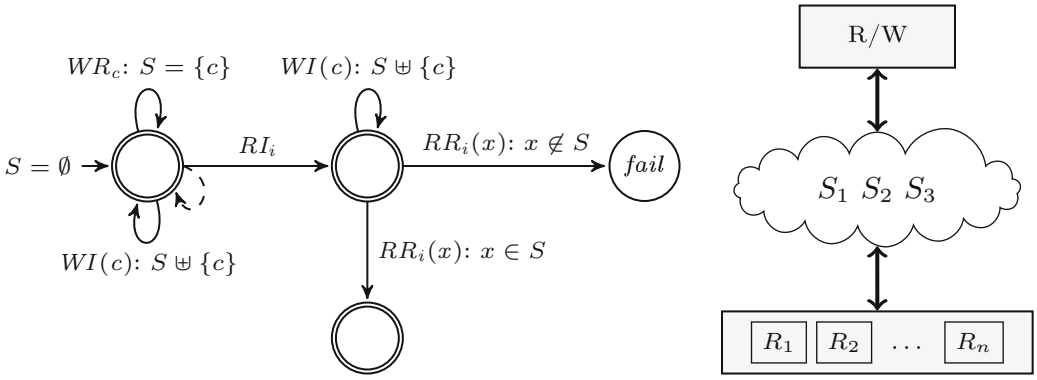


Fig. 9. Read-write automaton (left) and monitor deployment (right).

6 Test Case Execution

We have developed the QuoMBT test framework in order to perform model-based testing of quorum-based systems implemented using the Gorums framework. Also, we have implemented a client application and a distributed storage system which together with Gorums constitute the SUT. Figure 10 gives an overview of the testing framework comprised of CPN Tools and a test adapter. CPN Tools is used for modeling and generation of test cases and oracles (see Sects. 4 and 5). The generated

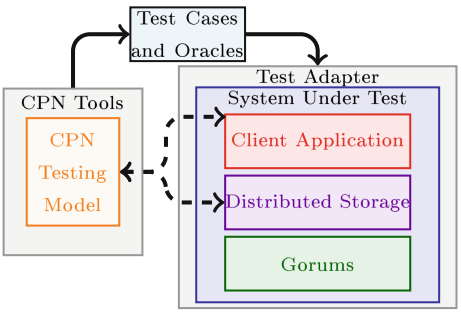


Fig. 10. QuoMBT testing framework.

test cases and oracles are written into XML files by CPN Tools, and then read by the test adapter. The reader of the test adapter feeds the test cases into the client application (test cases for quorum calls) and the distributed storage (test cases for quorum functions) implemented by the Gorums framework. Each test case is executed with the provided test values as inputs. The tester included in the test adapter compares the test oracle's output against the output of each test case in order to determine whether the test fails or succeeds. The test adapter is implemented in the Go programming language.

The reader in the test adapter can read XML files for unit tests of read and write quorum functions, and for system level tests involving quorum calls. The implementation of the reader uses Go's *encoding/xml* package, which makes it easy to define mappings between Go structs and XML elements. In order to map XML content into Go structs, each field of the Go struct has an associated XML tag, which is used by Go's XML decoder to identify the field to populate with content from the XML. We could have generated Go-based table-driven tests, which is already supported by the Go standard library. However, we chose to use an XML-based format for the generated test cases to enable reuse of the test generator across programming languages.

We have implemented the tester in the test adapter using the *testing* package provided by the Go standard library. This tester can start the implemented client application and execute generated test cases for the SUT. Go's testing infrastructure allows us to simply run the `go test` command to execute our generated tests, which will provide pass/fail information for each test case. In addition, this test infrastructure can also provide code coverage. When testing the distributed storage, we distinguish between quorum functions and quorum calls, because quorum functions are defined by developers when implementing their specific abstractions, whereas quorum calls are provided generally by the Gorums library. This separation also provides a modular approach to testing.

Our test adapter implements a Go-based tester for testing quorum functions, i.e., performing the unit tests. We simply iterate through the test cases obtained from the reader, invoking the ReadQF and WriteQF functions with the test values, and compare the results against the test oracles. The unit tests for read and write quorum functions can be performed without running any servers.

The system level tests require a set of running servers to test the complete system, including parts of the Gorums framework. When doing the system level tests involving quorum calls, the servers shown in Fig. 1 must be started first. Then, the test adapter starts a client so that it can execute the quorum calls. The test value, obtained from XML files, for each write quorum call is written to servers by calling the write quorum call, and for each read quorum call, the value returned by the servers will be captured by the tester to compare against the test oracle. For each write quorum call, the tests simply check if it returns an acknowledgment from servers.

The non-trivial part of the system test case execution is the concurrent and sequential executions of read and write quorum calls. For the detailed implementation of the storage involving quorum calls under test, the testing function for

quorum calls run through each test case read from the reader. For the run of each test case, the write and read quorum calls can be executed both sequentially and concurrently depending on the test driver used. For the sequential executions, the decision to execute write or read calls is made according to their sequences in the XML files generated by CPN Tools. For the concurrent executions of write and read quorum calls, the test execution makes use of go-routines provided by the Go programming language. Therefore, within each run of test cases, a write or read quorum call is executed based on their sequence in the XML files. Meanwhile, there may be other read calls that can be executed concurrently with the running write or read quorum call and this is then done in a separate go-routine. After executing each test case, the returned values of quorum calls are collected.

In order to obtain a test oracle for quorum calls which can be used in both state space-based and simulation-based test case generation, we use the automaton in Fig. 9 (left) to perform run-time verification of the Go implementation when executed on the system test cases derived from the CPN model. Specifically, our test adapter implements a *run-time monitor* corresponding to the automaton in order to keep track of the invoked and terminated write calls and thereby determining whether a value returned from a read call is permissible. Our test framework currently runs the client (the single writer and multiple readers) within a single Go process. This allows us to directly call into the monitor *before* the client sends the fan-out messages to servers, and *after* the quorum function returns the resulting quorum value, to check the result of the read request for plausibility against the permitted values specified above. This corresponds to monitoring *all* calls and returns in a particular deployment, i.e., correlating read calls and returns of the client in the system against those of the writer in the shaded area of Fig. 9 (right).

7 Experimental Evaluation

We now consider experimental evaluation of our model-based testing approach based on CPNs. In Sect. 7.1, we present in detail the results obtained for the distributed storage system. In Sect. 7.2 we summarise experimental results for an additional case study in which we have applied our approach to the Paxos consensus protocol for data replication. The main purpose of the Paxos case study is to demonstrate the generality of our approach and to show that it can be applied also to more complex examples of Gorums-based distribution systems. The library which we have developed for CPN Tools as part of this work to support test case generation is available via [19].

7.1 Results on Distributed Storage

To perform an evaluation of our model-based test case generation, we consider the code coverage obtained using different test drivers for concurrent and sequential execution of quorum calls in the client application. Our experimental evaluation comprises both successful scenarios and scenarios involving server failures

and programming errors. The toolchain of the Go language includes a code coverage tool which we have used to measure statement coverage.

Table 1 summarizes the experimental results obtained using different test drivers in which there are not server failures included. We consider the following test drivers: one read call (RD), one write call (WR), a read call followed by a write call (RD; WR), a write call followed by a read call (WR; RD), a read and a write call executed concurrently (WR||RD), a read and a write call executed concurrently and followed by a read call ((WR||RD); RD).

Table 1. Experimental results for distributed storage – successful scenarios.

Test driver		Test case generation					Test case execution (coverage in percentage)				
							System			Unit	
ID	Name	Nodes	Arcs	Time (seconds)	QC	QF	Gorums library	QCs		QFs	
								RD	WR	RD	WR
S1	RD	39	72	<1	1	3	24.6	84.4	0	100	0
S2	WR	39	72	<1	1	3	24.6	0	84.4	0	100
S3	RD; WR	254	543	<1	1	7	39.1	84.4	84.4	100	100
S4	WR; RD	254	543	<1	1	12	40.8	84.4	84.4	100	100
S5	WR||RD	1,549	4,379	1	6	17	40.8	84.4	84.4	100	100
S6	(WR||RD); RD	3,035	7,867	2	6	17	40.8	84.4	84.4	100	100

The table shows the number of nodes/arcs in the state space of the CPN model with the given test driver, the state space and test case generation time in seconds, the number of test cases generated for quorum calls (QC), the number of test cases generated for quorum functions (QF). For the test case execution, we show the code coverage (in percentage) that was obtained for the system level and unit tests. The results for successful execution scenarios show that the statement coverage for read (RD-QF) and write (WR-QF) quorum functions is 100% for both system and unit tests, as long as both read and write calls are involved. The statement coverage for read (RD-QC) and write (WR-QC) quorum calls is up to 84.4%. For the Gorums library as a whole, the statement coverage reaches 40.8%. It is worthwhile noting that the sizes of the state spaces considered are small. This is due to the fact that the CPN testing model describes the quorum-based system at a high level of abstraction which in turn is what makes the approach feasible.

The test cases considered above validates that the implementation of the distributed storage and the Gorums framework works correctly when there are no server failures. To further increase the code coverage and further evaluate our approach, we additionally evaluated the following aspects:

Programming errors. Gorums requires the developer to implement the quorum functions for the specific quorum-based system. To evaluate our ability

to detect programming errors in these function, we injected programming errors in the quorum functions for the distributed storage (see Algorithm 1) such that the requirement for having a quorum was incorrectly implemented.

Server and communication failures. Quorum-based systems are designed to tolerate server failures. To test the Gorums framework under such conditions, we consider the S6 driver from Table 1 and created a scenario in which first S6 is executed, then there is one or more server failures, and then S6 is repeated. A related scenario is that the client attempts to make quorum-calls before the servers have started.

Rerunning the test cases from Table 1 in the presence of programming errors resulted in the test cases not passing. This demonstrates our ability to detect programming errors in the quorum functions.

The server failures scenarios are handled in the test adapter by a component that can terminate any number of the servers when executing such test cases. Our test case execution showed that the distributed storage and the Gorums framework in a configuration with three servers are able to handle up to one server failure (as expected). The size of the state spaces generated for these scenarios ranged between 1,500 states (all servers failed) and 3,000 (no server failure). The total number of test cases ranged from 9 to 16.

The results for scenarios involving failures and programming errors show that the statement coverage for read (RD-QF) and write (WR-QF) quorum functions is still 100% for both system and unit tests. The statement coverage for read (RD-QC) and write (WR-QC) quorum calls is increased to 96.7%. For the Gorums library as a whole, the statement coverage is also increased to 52.3%. The reason for the lower coverage of the Gorums library is that it contains code generated by Gorums's code generator, and among them, various auxiliary functions that are never used by our current implementation. The total number of lines of code for the system under test is approximately 2100 lines, which include generated code by Gorums's code generator (around 1800 lines), server code (around 120 lines), client code (around 80 lines) and the code for quorum functions (around 60 lines).

As part of analyzing the results of the code coverage and experimenting with the test case generation, we also discovered a code path that was not covered. So we added an additional test that would cover this particular path. This involved passing **nil** as an argument to either the read (RD-QC) or write (WR-QC) quorum calls. The code path in question had recently been introduced to support a new feature in Gorums, but when the code path was exercised without activating its intended feature, the test case revealed that this code path had a bug causing the test client to panic. The bug has since been reported to the Gorums developers, and a fix has been implemented.

7.2 Results on the Paxos Consensus Protocol

To show that our approach is more generally applicable, we report on one additional case study which we have conducted with our model-based testing approach for CPNs and the support provided by QuoMBT. The example is an

implementation of the Paxos protocol using the Gorums framework. Paxos is a fault-tolerant consensus protocol that makes it possible to construct a replicated service using a group of server replicas. Paxos is considerably more complex than the distributed storage system, and each Paxos node (server replica) implements a proposer, an accepter, and a learner subsystem in addition to software components for failure and leader detection. Furthermore, three quorum calls (prepare, accept, and commit) are used in the implementation of the protocol. Due to the complexity of the Paxos protocol we have used simulation-based test case generation using up to 10 simulation runs to extract test cases.

Table 2 summarizes the experimental results obtained. The table shows the statement coverage obtained for the different subsystems of our Paxos implementation. Note that the Unit tests are for the quorum functions and hence not applicable for the other subsystems. The two numbers written below System tests and Unit tests gives the total number of test cases generated for 3 and 5 replica configurations, respectively. The test case generation for each configuration considered took less than 10 seconds, and the execution of each test case took less than one minute.

Table 2. Experimental results for test case generation and execution.

Subsystem	Component	System tests	Unit tests
		15/38	74/424
Gorums library		51.8%	-
Paxos core	Proposer	97.4%	-
	Acceptor	100.0%	-
	Failure detector	75.0%	-
	Leader detector	91.4%	-
	Replica	91.4%	-
Quorum calls	Prepare	83.9%	-
	Accept	83.9%	-
	Commit	83.9%	-
Quorum functions	Prepare	100.0%	90.0%
	Accept	100.0%	85.7%

The results show that the statement coverage of unit tests for Prepare and Accept quorum functions are up to 90% and 85.7%, respectively. For the system tests, the statement coverage for Prepare, Accept and Commit quorum calls reaches 83.9%, respectively; the results of statement coverage for Prepare and Accept quorum functions are up to 100%; for the Paxos implementation (Paxos core in the table), the Proposer module's statement coverage reaches 97.4%; the statement coverage of the Acceptor module is up to 100%; the statement coverages of the Failure Detector and Leader Detector modules reach 75.0% and 91.4%,

respectively; the statement coverage of the Paxos replica module reaches 91.4%; for the Gorums library as a whole, the highest statement coverage reaches 51.8%.

Similar to the distributed storage system, we obtain a high statement coverage for the Paxos implementation. In addition, using the generated test cases we identified several programming errors in the Paxos implementation which could then be fixed. To construct the CPN testing model for the Paxos protocol, we reused the modeling patterns for quorum functions and quorum calls developed for the distributed storage system. Furthermore, the test case generation algorithms were directly used without change to generate test cases for the Paxos protocol. The parts that had to be developed specifically for the Paxos protocol was the observation and detection functions, and parts of the formatting function used to generate the XML test case representation. Finally, parts of the test adapter had to be implemented to match the quorum calls and quorum functions that are specific for the Paxos implementation. This shows that significant parts of our approach can be used for other Gorum-based protocol implementations.

8 Related Work

Model-based testing is a large research area, and MBT approaches and tools have been developed based on a variety of modeling formalisms, including flowcharts, decision tables, finite-state machines, Petri Nets, state-charts, object-oriented models, and BPMN [13]. Saifan and Dingel's survey [20] provides a detailed description of how model-based testing is effective in testing different aspects of distributed systems, and it classifies model-based testing based on different criteria and compares several model-based testing tools for distributed systems based on this classification. The comparison, however, does not identify work that can be applied to systems that rely on a quorum system to achieve fault-tolerance.

The Gorums framework has only recently been developed, and hence there does not yet exist work that have considered model-based testing of this framework. Chubby [3] was one of the first implementations of Paxos that were deployed in a production environment, and thus were extensively tested. They highlight that at the time (2007), it was unrealistic to prove correct a real system of that size. Thus to achieve robustness, they adopted meticulous software engineering practices, and tested random sequences of network outages, message delays, timeouts, and process crashes. Using our CPN model and our generated tests, we aim to test many of the same attributes in a more systematic manner. Xiangdong et al. [10] applied a CPN-based simulation method on a quorum-based distributed storage system called Cassandra [1]. Cassandra is highly configurable, and the focus in their work was to find appropriate parameter settings to achieve the best performance. To this end, they developed a CPN-based simulator specifically for Cassandra, which allow tuning various system parameters such as cluster size, timeouts and read/write ratios, for their CPN models. In our work, we focus on using the CPN test models for generating test cases to perform both unit and system tests to the implementations of distributed systems.

The Integration and System Test Automation (ITSA) tool follows a CPN-based approach to MBT [28]. The tool can generate test code for a variety of languages including Java, C/C++, *C#* and HTML. Compared to the ITSA tool, our MBT approach is not tied to a particular programming language, since the test cases can be generated as an XML format, which can be read by any programming language. The ITSA tool also uses the state space of the testing model to generate and select test cases. To obtain concrete test cases with input data, the tool relies on a separate model-to-implementation mapping. In contrast, we obtain the input data for the quorum functions and calls directly from the data modeling contained in the CPN testing model. As a case study, the ISTA tool has been applied to an online shopping system. However, their approach does not appear to be suitable for testing complex distributed system protocols, since they do not handle concurrency and failures, which is at the core of our work in this paper.

Faria et al. [6] use timed event-driven CPNs to generate test code. They do not use CPNs as a direct interface to the user, but generate them from UML sequence diagrams. Their tool suite has a different focus in that they instrument a running system to observe the messages specified in the sequence diagrams. The toolset can only perform JUnit tests on Java-based applications, and it has not been used for unit and system testing of distributed systems. Liu et al. [18] has also proposed a CPN-based test generation approach. The approach requires defining a conformance testing-oriented CPN (CT-CPN) model and a PN-ioco relation specifying how an implementation conforms to its specifications. Their test case generation algorithm for the CT-CPN model is simulation-based. In our approach on the other hand, we can directly generate test cases using both state space-based and simulation-based test case generation for an existing implementation of the system under test. A model-based test generation technique based on CPNs is used by Daohua, Eckehard and Jan [27] to verify a module of a satellite-based train control system. They use CPN Tools to generate the reachability graph of the test model, and use state space analysis with CPN Tools to extract the expected output of each test case from the path of the reachability graph. However, their technique does not support simulation-based test case generation, which is of utmost importance for scalability.

Moreover, Watanabe and Kudoh [26] propose two different CPN-based test suite generation methods for concurrent systems. However, their methods do not directly address a particular way to derive a CPN testing model for a distributed system, nor do they analyze achieved code coverage. Wei et al. [29] describe two algorithms for generating test cases and test sequences from a CPN model. In their method, a CPN model of the system under test is created. This model is then used as input to their APCO algorithm to generate an initial set of test cases which can be converted to test sequences using their SPS algorithm. Then, the set of original test cases and the test sequences can be exported as XML formatted files. They demonstrated their MBT approach for testing a radio module in a centralized railway control system. In contrast to our approach, Wei et al. do not consider test scenarios with failures, do not handle concurrency, and their

approach has not been used to validate distributed systems. Finally, we discuss Farooq, Lam and Li's test sequence generation technique [7]. They derive a CPN model from a UML Activity Diagram, and use the derived model to generate test sequences. They demonstrate their approach on a fictional enterprise commerce system, describing the process of purchasing products online. In our MBT approach, we have designed a testing framework, consisting of the constructed CPN test models, test case generation algorithms and a test adapter, in order to enable the execution of the generated tests on real distributed systems.

9 Conclusions and Future Work

The main contribution of our work is an MBT approach that can be used for testing quorum-based distributed systems implemented using Gorums. Our approach includes modeling patterns, test case generation algorithms, and a test case execution infrastructure. As case studies, we have applied our approach to a distributed storage system and a Paxos implementation to illustrate and evaluate its applicability. The results are promising in that we have obtained high code coverage by considering both common case execution scenarios and failure scenarios. Furthermore, the results have been obtained with relatively simple test drivers and a small number of test cases. We have shown that in addition to obtaining results on code coverage, our generated unit and system tests are able to detect programming errors.

An important attribute of our approach is that the CPN testing model has been constructed such that it can serve as a basis for model-based testing of *other* quorum-based systems. This has been demonstrated by the application of our approach to the more complex Paxos consensus protocol. In particular, it is only the modeling of the quorum calls on the client and server side that are system dependent. To experiment with different quorum functions for a given quorum system, it is only the implementation of the quorum functions in Standard ML that needs to be changed. The state space and simulation-based test case generation approaches are independent of the particular quorum system under test. Our current solution uses the CPN model to generate test cases and record the correct response from the quorum function. The global monitor presented in Sect. 5 independently specifies safe behavior in the form of correct read calls.

The work presented in this paper opens up several directions of future work. We have obtained good coverage results on the quorum functions and quorum calls with the current testing model by considering both successful execution scenarios and scenarios involving server failures and programming errors. However, in order to increase coverage and consider more of Gorums's code paths, we need to test the quorum calls under additional failures scenarios and adverse conditions, such as network errors. This will require extensions to the model e.g. for generating timeouts, which in turn must be recorded in the test cases. This in turn will require extensions to the test adapter such that the environment can replay these events during test case execution.

Model-based testing can be used to test a system either by connecting a model (acting as a test driver) directly to an instance of the running system, or, as we do in this paper, generate test cases offline and execute these test cases against the system. The main challenge related to this, is how to handle non-determinism during test case execution. In our current approach, we have addressed this by using monitors known from the field of run-time verification. Instead of the automaton, a different formal specification logic for (distributed) systems could have been used, e.g. Scheffel and Schmitz's distributed temporal logic [21]. Their three-valued logic would allow us to adequately capture that the monitor has neither detected successful nor failed completion.

To further evaluate the generality of our modeling and test case generation approach, we need to apply it to additional quorum-based systems. For example, we can extend our current distributed storage to support multi-writer storages with multiple clients. This will challenge the limits of state space-based generation of test cases as was also demonstrated with the Paxos protocol. A future direction is to also extend our approach to be applicable also to non quorum-based distributed systems. In particular, it becomes important to investigate in more detail the test coverage that can be obtained with simulation versus the test case coverage that can be obtained with state spaces. We anticipate that this will motivate work into techniques for on-the-fly test case generation and test case selection during state space exploration.

References

1. Apache Software Foundation. Apache Cassandra. http://cassandra.apache.org
2. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message-passing systems. J. ACM **42**(1), 124–142 (1995)
3. Chandra, T.D., Griesemer, R., Redstone, J.: Paxos made live: an engineering perspective. In: Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC 2007, pp. 398–407. ACM (2007)
4. CPN Tools. CPN Tools. http://www.cpntools.org
5. CPN Testing Model for Gorum-based Distributed Storage, July 2018. http://home.hib.no/ansatte/lmkr/DistributedStorage.xml
6. Faria, J.P., Paiva, A.C.R.: A toolset for conformance testing against UML sequence diagrams based on event-driven colored Petri nets. Int. J. Softw. Tools Technol. Transfer **18**(3), 285–304 (2016)
7. Farooq, U., Lam, C.P., Li, H.: Towards automated test sequence generation. In: Australian Conference on Software Engineering (ASWEC 2008), pp. 441–450 (2008)
8. Google Inc. gRPC Remote Procedure Calls. http://www.grpc.io
9. Google Inc., Protocol Buffers. http://developers.google.com/protocol-buffers
10. Huang, X., Wang, J., Qiao, J., Zheng, L., Zhang, J., Wong, R.K.: Performance and replica consistency simulation for quorum-based NoSQL system cassandra. In: van der Aalst, W., Best, E. (eds.) PETRI NETS 2017. LNCS, vol. 10258, pp. 78–98. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57861-3_6
11. Jensen, K., Kristensen, L.M.: Coloured Petri nets: a graphical language for modelling and validation of concurrent systems. Commun. ACM **58**(6), 61–70 (2015)

12. Jepsen. Distributed Systems Safety Analysis. http://jepsen.io
13. Jorgensen, P.: The Craft of Model-Based Testing. CRC Press, Boca Raton (2017)
14. Kristensen, L.M., Simonsen, K.I.F.: Applications of coloured Petri nets for functional validation of protocol designs. In: Jensen, K., van der Aalst, W.M.P., Balbo, G., Koutny, M., Wolf, K. (eds.) Transactions on Petri Nets and Other Models of Concurrency VII. LNCS, vol. 7480, pp. 56–115. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38143-0_3
15. Kristensen, L.M., Veiset, V.: Transforming CPN models into code for TinyOS: a case study of the RPL protocol. In: Kordon, F., Moldt, D. (eds.) PETRI NETS 2016. LNCS, vol. 9698, pp. 135–154. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39086-4_10
16. Lamport, L.: The Part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998)
17. Lea, T.E., Jehl, L., Meling, H.: Towards new abstractions for implementing quorum-based systems. In: Proceedings of 37th IEEE International Conference on Distributed Computing Systems (ICDCS), pp. 2380–2385 (2017)
18. Liu, J., Ye, X., Li, J.: Colored Petri nets model based conformance test generation. In: IEEE Symposium on Computers and Communications (ISCC), pp. 967–970. IEEE (2011)
19. MBT/CPN. Repository, July 2018. https://github.com/selabhvl/mbtcpn.git
20. Saifan, A., Dingel, J.: Model-based testing of distributed systems. Technical report 548, School of Computing, Queen's University, Canada (2008)
21. Scheffel, T., Schmitz, M.: Three-valued asynchronous distributed runtime verification. In: Twelfth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE), pp. 52–61. IEEE (2014)
22. Stolz, V.: Temporal assertions with parametrized propositions. J. Logic Comput. **20**(3), 743–757 (2010)
23. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Softw. Test. Verif. Reliab. **22**, 297–312 (2012)
24. Vukolic, M.: Quorum Systems: With Applications to Storage and Consensus. Morgan and Claypool, San Rafael (2012)
25. Wang, R., Kristensen, L.M., Meling, H., Stolz, V.: Application of model-based testing on a quorum-based distributed storage. In: Proceedings of PNSE 2017. CEUR Workshop Proceedings, vol. 1846, pp. 177–196 (2017)
26. Watanabe, H., Kudoh, T.: Test suite generation methods for concurrent systems based on coloured Petri nets. In: Software Engineering Conference, pp. 242–251. IEEE (1995)
27. Wu, D., Schnieder, E., Krause, J.: Model-based test generation techniques verifying the on-board module of a satellite-based train control system model. In: 2013 IEEE International Conference on Intelligent Rail Transportation Proceedings, pp. 274–279, August 2013
28. Xu, D.: A tool for automated test code generation from high-level Petri nets. In: Kristensen, L.M., Petrucci, L. (eds.) PETRI NETS 2011. LNCS, vol. 6709, pp. 308–317. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21834-7_17
29. Zheng, W., Liang, C., Wang, R., Kong, W.: Automated test approach based on all paths covered optimal algorithm and sequence priority selected algorithm. IEEE Trans. Intell. Transp. Syst. **15**(6), 2551–2560 (2014)

PAPER B

AUTOMATED TEST CASE GENERATION FOR THE PAXOS SINGLE-DECREE PROTOCOL USING A COLOURED PETRI NET MODEL

R. Wang, L. M. Kristensen, H. Meling, and V. Stolz.

In Journal of Logical and Algebraic Methods in Programming, volume 104, pages 254–273, Elsevier Ltd, 2019.

Journal of Logical and Algebraic Methods in Programming 104 (2019) 254–273

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Automated test case generation for the Paxos single-decree protocol using a Coloured Petri Net model

Rui Wang [a,*], Lars Michael Kristensen [a], Hein Meling [b], Volker Stolz [a]

[a] Department of Computing, Mathematics, and Physics, Western Norway University of Applied Sciences, Norway
[b] Department of Electrical Engineering and Computer Science, University of Stavanger, Norway

ARTICLE INFO

Article history:
Received 27 March 2018
Received in revised form 10 December 2018
Accepted 13 February 2019
Available online 14 February 2019

Keywords:
Coloured Petri Nets
Distributed systems
Model-based testing

ABSTRACT

Implementing test suites for distributed software systems is a complex and time-consuming task due to the number of test cases that need to be considered in order to obtain high coverage. We show how a formal Coloured Petri Net model can be used to automatically generate a suite of test cases for the Paxos distributed consensus protocol. The test cases cover both normal operation of the protocol as well as failure injection. To evaluate our model-based testing approach, we have implemented the Paxos protocol in the Go programming language using the quorum abstractions provided by the Gorums framework. Our experimental evaluation shows that we obtain high code coverage for our Paxos implementation using the automatically generated test cases.

1. Introduction

Systematic testing is an important activity in software development. This is especially important for fault-tolerant distributed systems, because they are notoriously difficult to implement correctly [1]. The reason for this difficulty is that such systems must cope with both concurrency and failures, e.g. due to crashes and network partitions. Distributed systems therefore employ protocols with complex logic to tolerate individual component failures without causing service disruption for users. Testing approaches and programming abstractions that can be used to systematically test and simplify the implementation of protocols for distributed systems are therefore important.

Model-based testing (MBT) [2] has emerged as a powerful approach for testing software, and as part of our ongoing research effort, we are investigating the application of MBT on protocols for state machine replication (SMR). SMR is a core technique for developing fault-tolerant distributed systems that can tolerate a bounded number of server failures. In MBT, we construct a model of the system under test (SUT) and its environment, in order to generate test cases. The goal of MBT is validation and error-detection by finding observable differences between the behavior of the implementation and the intended behavior of the SUT, as defined by the model. A test case consists of inputs to the SUT and the expected output, which determines whether the execution of the test was successful or failed. Finally, it involves implementing a test adaptor that can be used to embed the SUT, enabling the test cases to be executed against the SUT, and their output compared against the expected output.

Coloured Petri Nets (CPNs) [3] are a formal modeling language that can model distributed systems combining Petri Nets and the Standard ML programming language. Petri Nets provide the foundation for modeling concurrency, synchronization,

* Corresponding author.
E-mail addresses: rwa@hvl.no (R. Wang), lmkr@hvl.no (L.M. Kristensen), hein.meling@uis.no (H. Meling), vsto@hvl.no (V. Stolz).

https://doi.org/10.1016/j.jlamp.2019.02.004
2352-2208/

communication, and resource sharing, while Standard ML provides the primitives for compact data modeling and sequential computations. Construction, simulation, validation, and verification of CPN models are supported by CPN Tools [4]. CPNs and CPN Tools have been widely used for modeling, validation, and verification of distributed systems protocols [5], but their application in software testing has only been explored to a limited extent [6–8]. Recently, CPNs have been explored in the context of automated code generation to obtain an implementation of a modeled system [9]. Even though the automated code generation is applied to obtain such an implementation of the modeled system, it is seldom that the correctness of the model-to-text transformations and their implementation can be formally proved. Thus, it is also an important task in the engineering of distributed systems to comprehensively test the implementation. Therefore, we have developed the MBT/CPN library [10] that extends CPN Tools with the support for model-based test case generation. The reason we chose CPNs as the foundation for our testing approach is that CPNs have a strong track record for modeling distributed systems and are able to create parametric models and perform model validation. Moreover, CPNs also have mature tools to support both simulation and state space exploration, which is important for implementing our approach and for our practical experiments.

The contribution of this paper is the application of CPNs and the MBT/CPN library [10] for model-based testing of an implementation of Paxos [11]. Paxos is a fault-tolerant consensus protocol that makes it possible to construct a replicated service, or SMR, using a group of server replicas. Paxos is an important foundational building block, and a whole family of Paxos-based protocols have been developed [12–15], focusing on different attributes such as latency and throughput. Moreover, Paxos is also the basis for many production systems such as Google's Chubby [16] and Spanner [17], and Amazon Web Services [18]. However, Paxos is also known for being difficult to understand and implement correctly [19]. The main aim of our work has been to develop a practically-oriented approach that narrows the gap between the provably correct in theory, and a correct implementation in practice. We use finite-state model checking to automatically validate the correctness of small configurations of the CPN model used for test case generation. This increases confidence in the test cases that are then subsequently extracted from running a set of simulations of the CPN model. The use of simulation to extract test cases (which are then executed against the SUT) ensures scalability of our approach. It also means that our approach (in general) only tests the SUT against a subset of the behaviors specified by the CPN model. As such our approach should be seen as aimed at validating an implementation and detecting implementation errors.

A secondary contribution is an implementation of the single-decree Paxos protocol that is especially amenable to testing. Single-decree Paxos allows a collection of servers to operate as a coherent group and to agree on a common value, while tolerating the failure of some of its members. The implementation, written in Go, takes advantage of quorum abstractions provided by the Gorums middleware library [20]. These abstractions include the ability to perform invocations on a set of server replicas, and collect, analyze, and combine a quorum of replies into a single representative reply to be used in the next protocol step. These abstractions also help to shield the programmer from having to explicitly deal with low-level communication and error handling.

The paper is organized as follows. §2 introduces the Paxos consensus protocol and gives an overview of the constructed CPN model, while §3 provides detailed models of the various Paxos agents. §4 gives an overview of Gorums and its abstractions, and outlines our Gorums-based implementation of Paxos. §5 presents our testing approach, and our test adapter developed to execute the test cases generated from the CPN model. §6 discusses test case generation and our experimental results obtained using CPN Tools and the MBT/CPN testing library. §7 discusses related work. Finally, §8 concludes the paper and discusses future work. The reader is assumed to be familiar with the basic syntactical and semantical concepts of Petri Nets (places, transitions, tokens, and transition enabling and occurrence). The CPN model and Paxos implementation presented herein are only partial. The full details of the CPN model are available via [21].

2. The Paxos consensus protocol and CPN model overview

The objective of a distributed *consensus* algorithm such as Paxos is to have a single value chosen among those proposed, while the *safety* (S) and *liveness* (L) properties [22,23] below are upheld with a *correct replica* being a replica that does not fail:

S1 Only a proposed value may be chosen.
S2 Only a single value is chosen.
S3 Only a chosen value may be learned by a correct replica.
L1 Some proposed value is eventually chosen.
L2 Once a value is chosen, correct replicas eventually learn it.

Note that the definition permits multiple values to be proposed for consensus. An algorithm satisfying the above safety properties is considered safe in that all replicas that learn the chosen value remain consistent with each other. However, we note that distributed consensus is impossible in an asynchronous system model [24]. Therefore, to satisfy liveness, periods of synchrony are required.

The single-decree Paxos consensus protocol can be used by a distributed application, in which the Paxos replicas need to agree on a single common value among potentially many input values. We assume that the input values are sent to the Paxos replicas from one or more clients, and then the decided output value is returned to these clients.

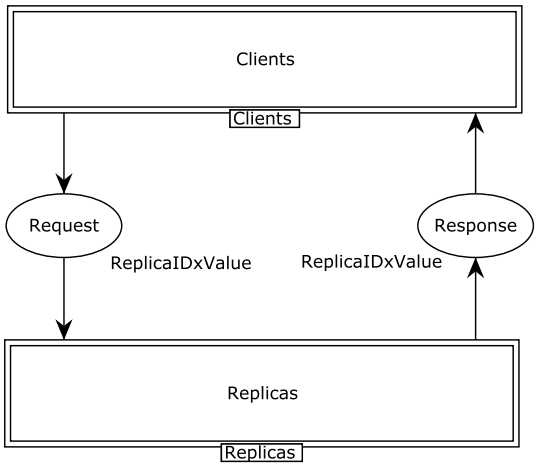


Fig. 1. Top-level CPN module for Paxos.

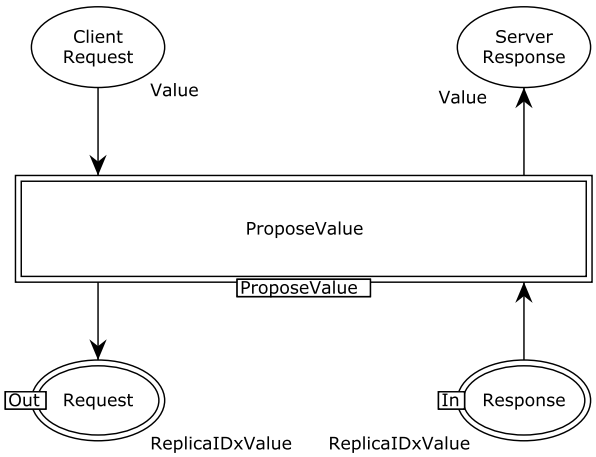


Fig. 2. The Clients CPN module.

The constructed CPN model of the single-decree Paxos protocol is comprised of 23 hierarchically organized modules. Fig. 1 shows the top-level module of the CPN model consisting of two substitution transitions (drawn as rectangles with a double border) connected by the two places Request and Response. The name of the submodule associated with each substitution transition is written in the name-tag positioned next to the substitution transition. The substitution transition Clients and its associated submodule Clients are modeling the behavior of the clients that may propose values to be chosen. The substitution transition Replicas and its associated submodule are modeling the behavior of the distributed replicas executing the Paxos protocol in order to reach consensus on a value proposed by the clients. The client sends a request to the Paxos replicas by putting a token on the place Request and then waits for the decided response value to be returned as a token on place Response.

Fig. 2 shows the submodule of the Client substitution transition. The port places Request and Response are associated with the identically named socket places in Fig. 1. This means that any tokens added to or removed from these places by transitions in the ProposeValue module will be reflected in the top-level module. The submodule of the ProposeValue substitution transition models the behavior of sending a client request value for consensus to the Paxos replicas.

Paxos [11,22] is often described in terms of three separate agent roles: *proposers* that can propose values, *acceptors* that accept a value among those proposed, and *learners* that learn the chosen value. A Paxos replica may take on multiple roles: in a typical configuration (and also in the CPN model), all replicas play all roles. Paxos is safe for any number of crash failures, and can make progress with up to f crash failures, given $n = 2f + 1$ acceptors.

Fig. 3 shows the Replicas module which is the submodule of the substitution transition Replicas in Fig. 1. The module has a substitution transition for each Paxos agent connected by socket places to model the communication between the different agents. The detailed behaviors of the agents are then modeled in the submodule of the substitution transitions. The Replicas module has been constructed such that we can configure any number of replicas, each encapsulating the three Paxos agents, without modifying the net-structure. This allows us to easily generate test cases for differently sized Paxos configurations.

The Paxos protocol operates in *rounds*, which refer to a set of semantically related messages that may or may not conclude the consensus protocol. We say that the protocol solves consensus in some round. Due to asynchrony and failures,

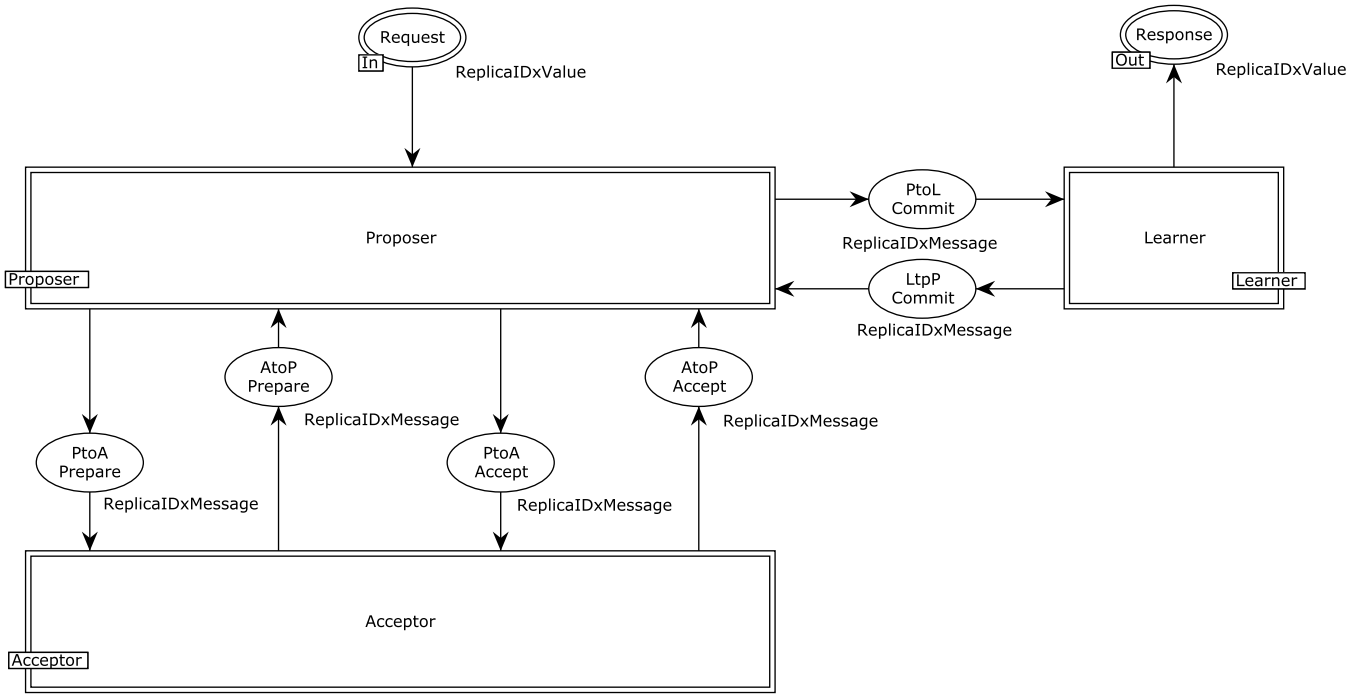


Fig. 3. The single-decree Replicas module.

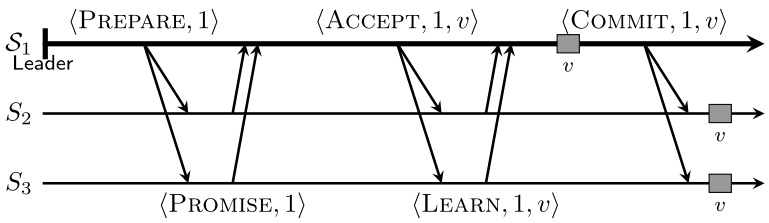


Fig. 4. The single-decree Paxos consensus protocol with three phases.

a consensus protocol may need to run several rounds to solve consensus. When describing the protocol, we use the variables *rnd*, *crnd*, and *vrnd* to denote round number, current round, and voted in round. Every round is associated with a single proposer, which is the *leader* for that round. Other proposers can start new (higher) rounds concurrently by sending a ⟨Prepare⟩ message to acceptors, to collect ⟨Promise⟩s from the acceptors to follow a new proposer. This is essential for Paxos to make progress in case the current leader goes mute. Every round runs in three phases:

1. A proposer sends a ⟨Prepare⟩ message to the acceptors and collects at least $f + 1$ ⟨Promise⟩ messages;
2. the proposer then sends ⟨Accept⟩ messages for some value v to the acceptors, who respond by sending ⟨Learn⟩ messages back to the proposer acknowledging the value v;
3. the proposer sends the decided value in ⟨Commit⟩ messages to learners.

The common case execution of the three phases is shown in Fig. 4. The first number in each message is the $rnd = 1$, and v is the value that the proposer wants the acceptors to choose. The gray boxes labeled v represent the execution of a state machine command derived from the decided value v. While not shown in the figure, each replica has instances of each of the Paxos agents. The communication between the different Paxos agents has been modeled based on the quorum abstractions provided by the Gorums framework [20], which we discuss in §4. Specifically, the communication takes the form of quorum calls, one for each of the Paxos phases: Prepare, Accept, and Commit.

The value v to choose is the value with the highest round among those provided in the ⟨Promise⟩ messages, or if no votes are provided in the ⟨Promise⟩ messages, any value can be chosen by the proposer; this would typically be a value that the proposer received from a client. In Paxos, acceptors are said to have *chosen* a value v, if a majority of acceptors have voted for v in the same round. Once a value has been chosen by acceptors in a round, no other value can be chosen in any other round. However, if there is no majority of acceptors that have voted for v, then the acceptors may vote for different values in other rounds. Since rounds execute concurrently, there is no guarantee of progress even if there are no failures or message loss. Therefore, Paxos typically relies on an eventual leader detection protocol, often implemented from

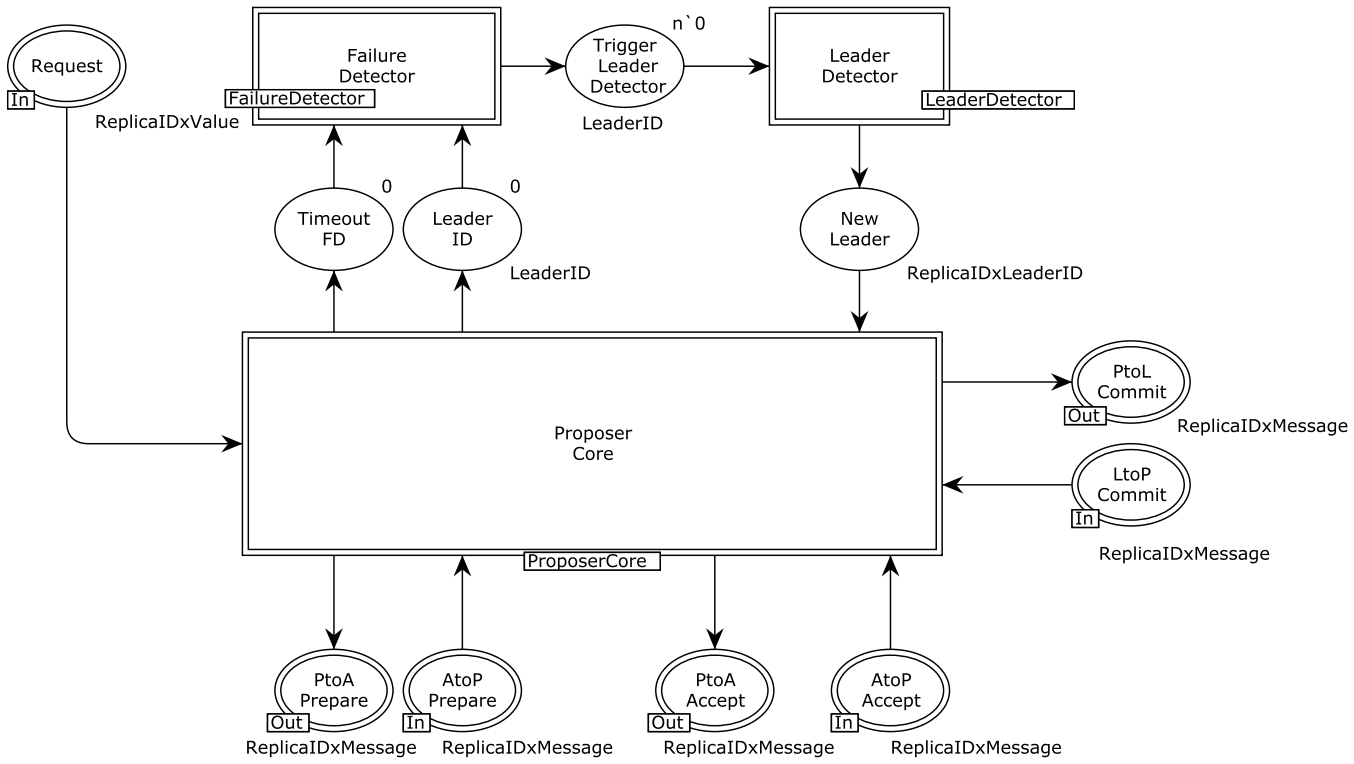


Fig. 5. The Proposer module.

the Ω failure detector [25]. While Ω may be inaccurate for some time, eventually it makes correct proposers agree on which proposer is the leader. Using Ω, and under the assumption that $n \geq 2f + 1$ acceptors, Paxos satisfies Properties L1 and L2.

3. Modeling the Paxos agents

This section presents the behavioral modeling of the Paxos agents. One of the Proposers is designated as a leader and proposes the client request value for consensus. The Acceptors choose the consensus value among those proposed, and the Learner of each replica learns the decided value. Once a value has been learned by a Learner, a response may be sent by this Learner to the client. This response is presented as a token on the port place Response.

3.1. Proposers

The submodule of the Proposer substitution transition is shown in Fig. 5. It contains three substitution transitions: LeaderDetector, FailureDetector, and ProposerCore. The Proposer of each replica receives the client request (presented as a token on the port place Request) for consensus, sent from the submodule of the Clients substitution transition.

In Paxos, one of the Proposers is responsible for driving the consensus process, namely the *leader*. However, due to the asynchronous nature of the environment in which we are operating, we may have that many Proposers believe they are the leader, thus the Paxos protocol can only guarantee progress if one of them is eventually chosen. Therefore, the objective of the first phase of Paxos is to obtain permissions from the Acceptors that a particular Proposer can serve as the leader. However, to be able to detect if a new proposer should initiate the first phase, we use the LeaderDetector substitution transition which has a submodule to pick a leader among the Proposers. This submodule is informed about failures from the failure detector. The FailureDetector substitution transition has a submodule that can detect the failure of any of the Proposers. Then, another leader can be selected by the submodule of the LeaderDetector substitution transition and it can take over the leadership by starting the first phase of Paxos with a higher round number than previous leaders.

Paxos uses round numbers to rank replicas, and each replica has a unique set of round numbers. More specifically, each round is assigned to a single proposer. The choice of the proposer for round i is determined by a deterministic mapping $p : B \rightarrow P$, where B is the set of round numbers and P is the set of proposers. In this paper, we assume that B is the set of natural numbers, and that proposers have assigned identities $0, 1, \ldots, |P| - 1$, where $|P| = n$. Then, we can choose mapping p such that $p(i) = i \pmod{|P|}$.

A client request presented as a token value on the port place Request will be sent to the ProposerCore, waiting to be handled by the leader as can be seen from Fig. 5.

The submodule of the ProposerCore substitution transition is shown in Fig. 6 and models the internal behavior of the Proposer. In this module, the InitProposer substitution transition has a submodule to initialize Proposers, obtain a new

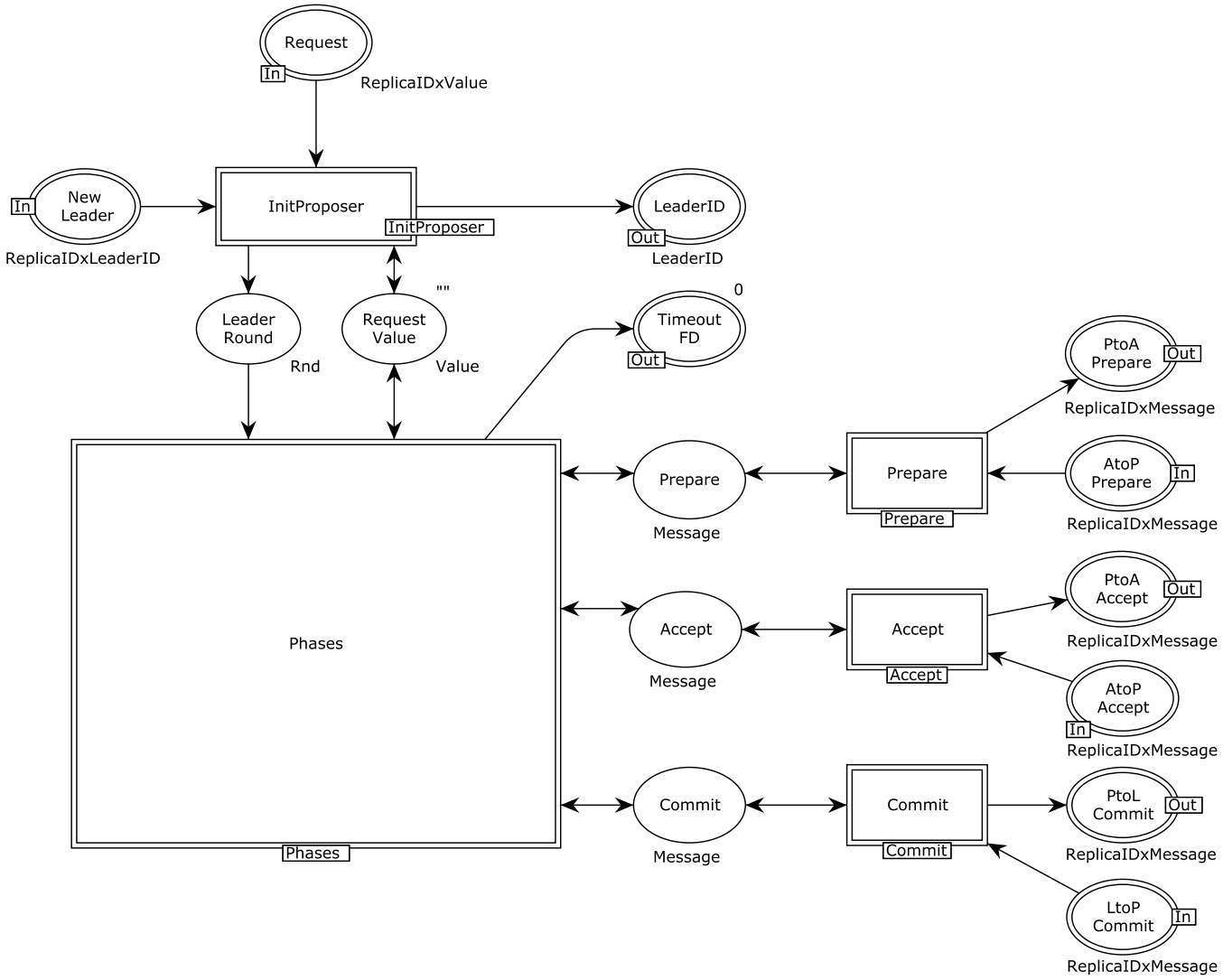


Fig. 6. The ProposerCore module.

leader, and receive a client request for consensus. Then, the value of the current round number of the leader and the value of the received client request will be presented on the port places as tokens, respectively. These tokens will be handled by the submodule of the Phases substitution transition.

A successful round of the single-decree Paxos protocol has three phases, modeled by submodules of the three substitution transitions shown in Fig. 7. The first phase involves two types of messages known as the ⟨Prepare⟩ and ⟨Promise⟩ messages. The leader candidate creates a ⟨Prepare⟩ message with its current round number and invokes a Prepare quorum call. This is modeled by the submodule of the Prepare substitution transition, shown in Fig. 6, which sends the ⟨Prepare⟩ message to Acceptors in order to propose itself to be a leader. After the Acceptors receive the ⟨Prepare⟩ message, and if they accepted it, then each Acceptor returns back a ⟨Promise⟩ message to the leader candidate by the Prepare quorum call. This is modeled by the submodule of the Acceptor substitution transition shown in Fig. 3. When the leader candidate receives enough ⟨Promise⟩ messages from Acceptors (obtain a quorum), then the first phase is finished, which means the leader candidate now can become a leader, and propose the client request to Acceptors for consensus.

In the second phase, the leader creates an ⟨Accept⟩ message with its current round number, *crnd*, and the value *v* obtained from the client request, and invokes the Accept quorum call, modeled by the submodule of the Accept substitution transition, shown in Fig. 6. This quorum call sends the ⟨Accept⟩ message to the Acceptors, requesting them to vote for consensus value *v*. Upon receiving an ⟨Accept⟩ message whose round number is greater or equal to the Acceptor's round number, the Acceptor will return a ⟨Learn⟩ message to the leader. Once the leader has received a quorum of ⟨Accept⟩ messages from Acceptors, the second phase is done. For the third phase, the leader invokes the Commit quorum call on the Learners, as modeled by the submodule of the Commit substitution transition, shown in Fig. 6. This enables the Learners to learn the chosen consensus value and can send it to the client.

Fig. 8 shows the submodule of the PhaseOne substitution transition. The leader uses its current round number to create a ⟨Prepare, 0, *crnd*⟩ message by triggering the transition SendPrepareMessage so that this message can be placed on the port

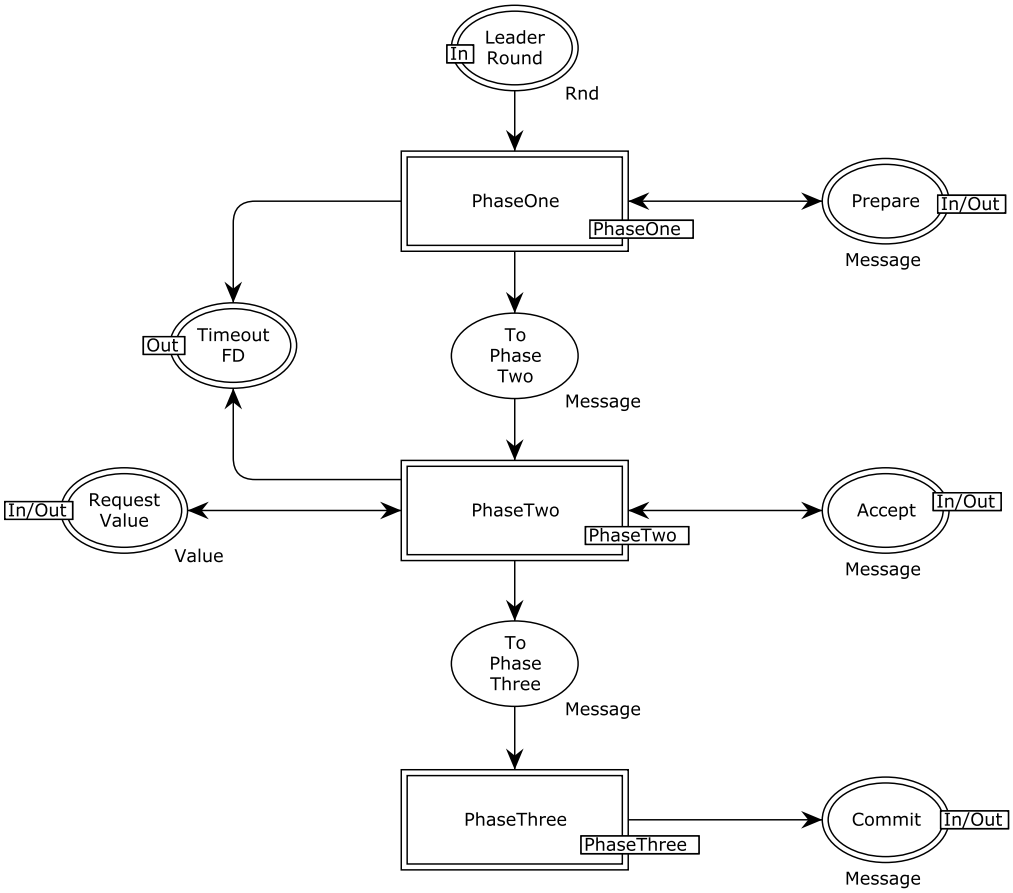


Fig. 7. The Phases module.

place Prepare as a token to invoke the Prepare quorum call. This quorum call returns a ⟨Promise, *cid*, *rnd*, (*vrnd*, *vvalue*)⟩ message as we already discussed, where *cid* is the call ID (initialized as 0 in ⟨Prepare⟩ message); *rnd* is the round number confirmed by the Acceptor; *vrnd* is the most recent round in which the Acceptor voted, and *vvalue* is the consensus value it voted for. The place FDControl provides an upper bound on the number of timeouts/failures in our test cases. This place is not part of the CPN model for single-decree Paxos, but is used to control the test environment. If no timeout occurs, and the leader obtained a quorum of ⟨Promise⟩ messages, the second phase can start. The place FailedReplica is used to collect the identity of failed replicas, which we use in §6 for validation of the model. The second and third phases are modeled in a similar manner as the first phase, and we do not include them here.

After the ⟨Promise⟩ message returns, a timeout could happen to trigger the failure detector when the ProcessPromiseMessage transition occurs. This is used to capture scenarios where a failure of any replica occurs. Such failure is modeled as an event that may occur after a quorum has been obtained for the quorum call, which, in this case, is represented as a token of the ⟨Promise⟩ message appearing on the port place Prepare. At this stage, an occurrence of the ProcessPromiseMessage transition (Fig. 8) may result in a timeout modeled by the creation of a token on the port place TimeoutFD signaling that a failure has occurred. We may then have a finite sequence of transition occurrences for the accomplishment of the Prepare quorum call (in this case) and for finishing the remaining transitions in the submodule of the LeaderDetector substitution transition. After this, the transitions for leader detection in the submodule of the FailureDetector substitution transition will occur as they are given higher priority compared to other transitions in the model. This ensures that the execution of the failure detector cannot be forever postponed and the current leader ID (round number) for this failed round is obtained, which then causes the execution of the leader detector to elect a new leader. The fact that the failure detector will be executed in a finite number of steps from when a failure has occurred, restricts the behavior of the model and in turn implies that the model satisfies properties L1 (a proposed value is eventually chosen) and L2 (that correct replicas eventually learns the chosen value).

3.2. Acceptors

This section details the model for the acceptors. Fig. 9 shows the Acceptor module. The submodule consists of HandlePrepare and HandleAccept substitution transitions. The former handles ⟨Prepare⟩ messages sent by the submodule of the Proposer substitution transition shown in Fig. 3 through the port place PtoAPrepare. The latter similarly handles ⟨Accept⟩ messages also sent by the Proposer through port place PtoAAccept.

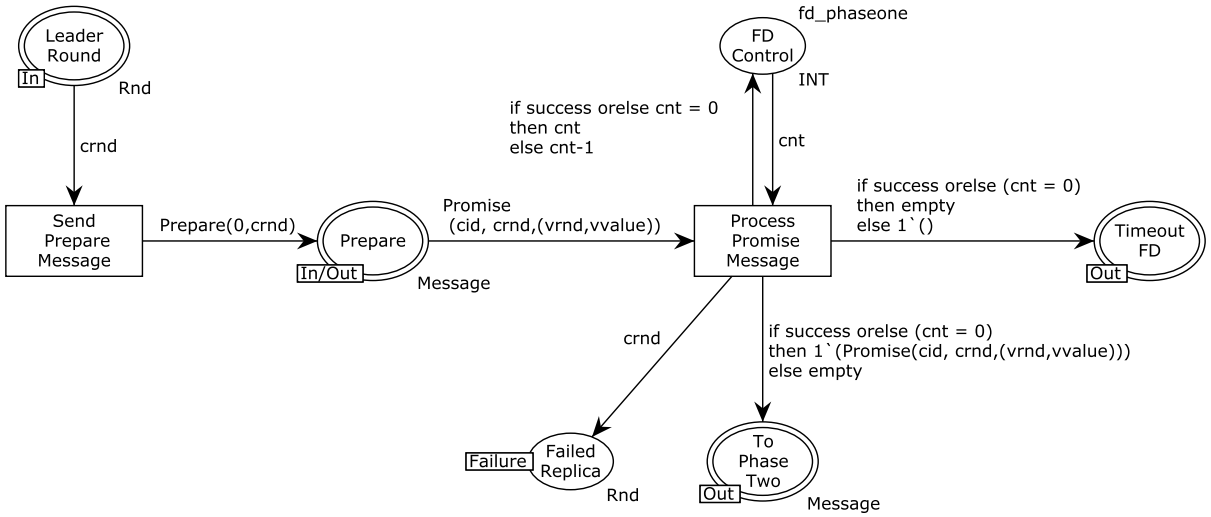


Fig. 8. The PhaseOne module.

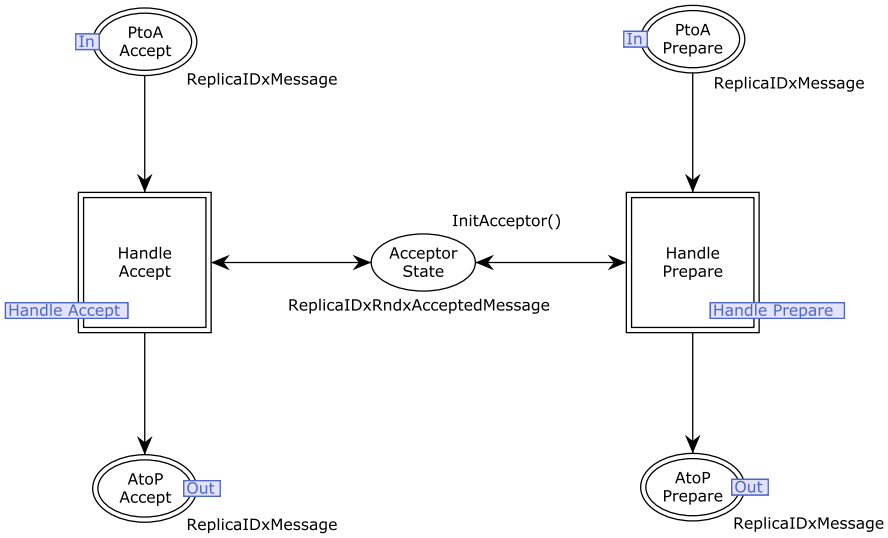


Fig. 9. The Acceptor module.

The AcceptorState place represents the state of each Acceptor. It is initialized with each replica's ID, round number $rnd = 0$, last voted round $vrnd = 0$, and voted value $vvalue = \varepsilon$ (empty string).

The submodule of the HandlePrepare substitution transition is shown in Fig. 10. The HandlePrepare transition handles ⟨PREPARE⟩ messages sent by the Proposer. If the *crnd* of the ⟨PREPARE⟩ message is higher than the Acceptor's *rnd*, then the token placed on the AcceptorState port place is updated accordingly, and a new token for the ⟨PROMISE⟩ message can be placed on the AtoPPrepare port place according to the expression of the arc connecting the HandlePrepare transition and AtoPPrepare place. We do not show the submodule of the HandleAccept substitution transition as it is similar to HandlePrepare. The main difference is that it updates the triplet $(rnd, vrnd, vvalue)$ in the AcceptorState port place, and places a ⟨LEARN⟩ message on the AtoPAccept place.

3.3. Learners

Finally, we discuss the Learner substitution transition shown in Fig. 3, which has a submodule with a single HandleCommit substitution transition, as shown in Fig. 11. This submodule handles the ⟨LEARN⟩ message sent by the Proposer, checking that a quorum of learn messages have been received before returning the decided consensus value to the client by placing a token on the Response port place. This behavior is modeled by the submodule of the HandleCommit substitution transition, shown in Fig. 12.

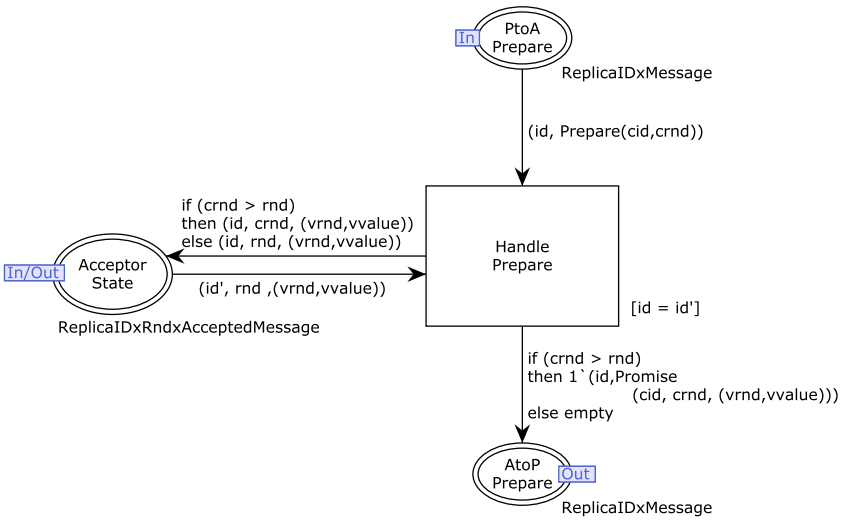


Fig. 10. The HandlePrepare module.

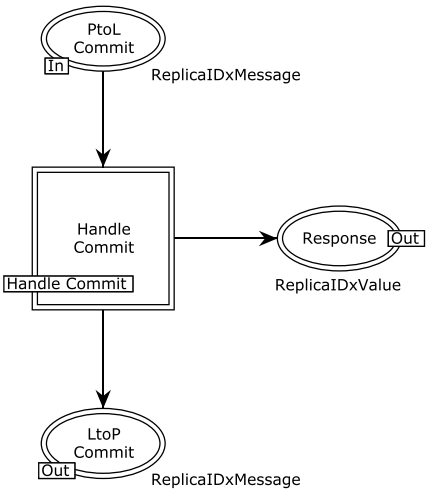


Fig. 11. The Learner module.

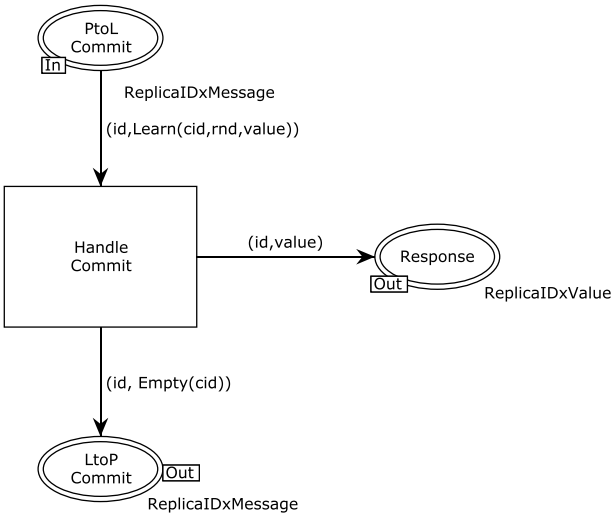


Fig. 12. The HandleCommit module.

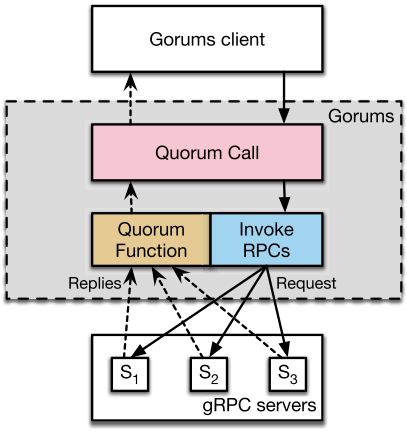


Fig. 13. Gorums abstractions.

4. Gorums and single-decree Paxos implementation

Gorums [20] is a framework for implementing quorum-based distributed systems. This section describes Gorums and how we use it to implement single-decree Paxos [11]. Our goal here is to provide a Paxos implementation that is amenable to testing based on the CPN model in §3, and in §5 we describe our testing approach.

4.1. Gorums abstractions

Gorums is a library whose goal is to alleviate the development effort for building advanced distributed algorithms, such as Paxos [11] and distributed storage [26,27]. These algorithms are commonly used to implement replicated services, and they rely on a quorum system [28] to achieve fault tolerance. In a quorum system, such as Paxos, protocol replicas need to exchange and update information about each other's state. However, to ensure consistency, a replica must contact a quorum, e.g. a majority of the replicas. In this way, a system can provide service despite the failure of individual replicas. However, communicating with and handling replies from sets of replicas often complicate the protocol implementations.

To reduce this complexity, Gorums provides three core abstractions: (a) configurations that group a set of replicas to hide the existence of individual replicas, (b) a flexible and simple quorum call abstraction, which is used to communicate with a configuration, i.e. a set of replicas, and to collect their responses, and (c) a quorum function abstraction which is used to process responses. These abstractions can help to simplify the main control flow of protocol implementations, as we illustrate later in this section.

Fig. 13 illustrates the interplay between the main abstractions provided by Gorums. Gorums consists of a runtime library and code generator that extends the gRPC [29] remote procedure call library from Google. Specifically, Gorums allows clients to invoke a quorum call, i.e. a set of RPCs, on a group of servers, and to collect their replies. The replies are processed by a quorum function to determine if a quorum has been received. Note that the quorum function is invoked every time a new reply is received at the client, to evaluate whether or not the received set of replies constitutes a quorum.

Protocol developers using Gorums can specify RPC service methods using `protobuf` [30], and from this specification, Gorums's code generator will produce code to facilitate quorum calls and collection of replies. Each quorum call method must provide a user-defined quorum function that Gorums will invoke to determine if a quorum has been received for that specific quorum call. In addition, the quorum function will also provide a single reply value, based on a coalescing of the received reply values from the different server replicas. This coalesced reply value is then returned to the client as the result of its quorum call. That is, the invoking client does not see the individual replies.

The quorum functions for a specific protocol implementation must follow a well-defined interface generated by Gorums. These only require a set of reply values as input and a return of a single reply value together with a boolean quorum decision. Hence, quorum functions can easily be tested using manually written unit tests. However, some quorum functions involve complex logic, and their input and output domains may be large, and so generating test cases from a model, provide significant benefit for verifying their correctness.

A quorum call is implemented by a set of RPCs, invoked at different servers, and so must consider different interleavings due to invocations by different clients. Hence, using model-based testing we can produce sequences of interleavings aimed at finding bugs in the server-side implementations of the RPC methods and also in the Gorums runtime system.

Fig. 14 gives the Prepare quorum call module of the Prepare substitution transition in Fig. 6. This module models the behavior of the quorum call and quorum function abstractions provided by Gorums for sending the ⟨PREPARE⟩ messages from a Proposer (leader) to Acceptors when the transition SendPrepareMessages occurs. Then, after such ⟨PREPARE⟩ messages are handled by Acceptors, the replied ⟨PROMISE⟩ messages from Acceptors can be processed when the transition ApplyPrepareQF occurs, which models the behavior of the Prepare quorum function. The logic to implement such quorum function will be discussed in §4.2.

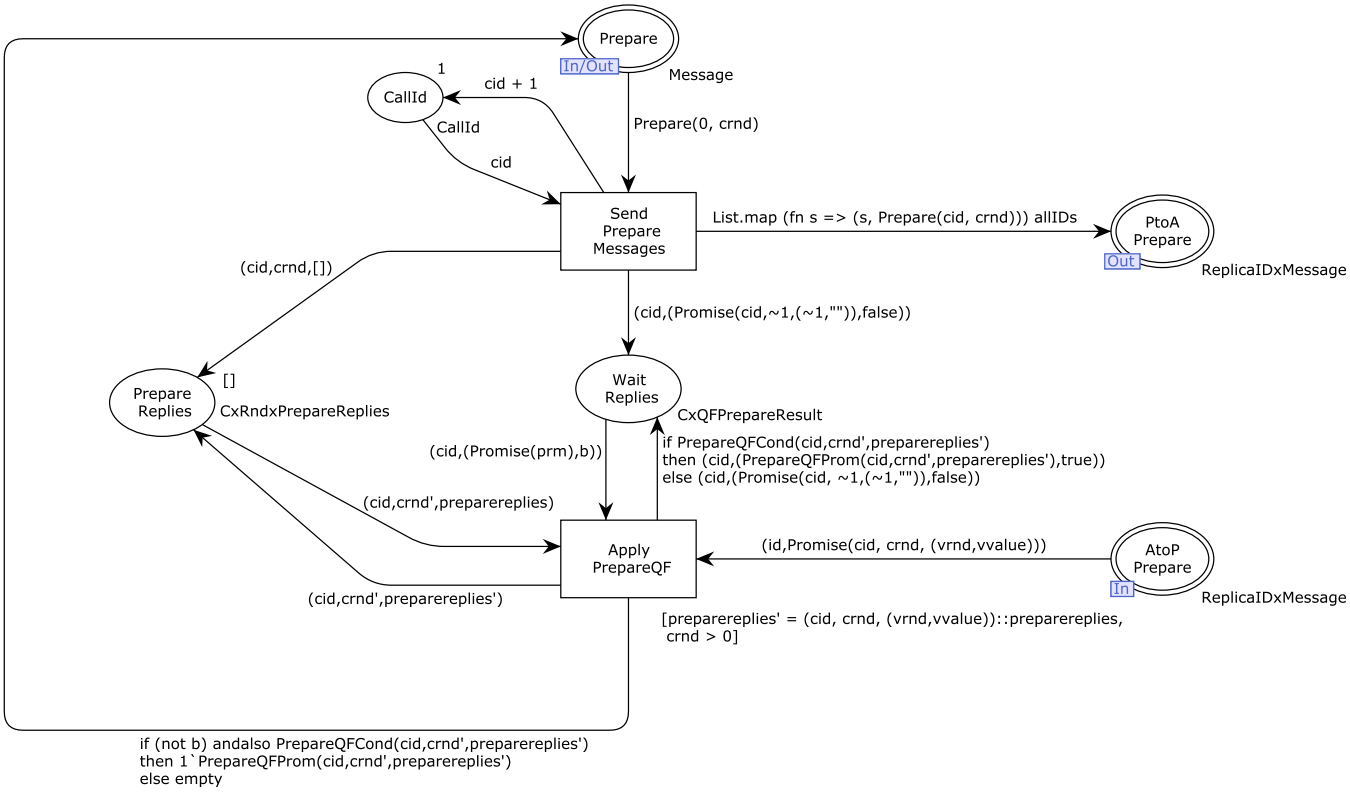


Fig. 14. The Prepare quorum call module.

```
type SinglePaxosServer interface {
    Prepare(context.Context, *PrepareMsg) (*PromiseMsg, error)
    Accept(context.Context, *AcceptMsg) (*LearnMsg, error)
    Commit(context.Context, *LearnMsg) (*Empty, error)
    ClientHandle(context.Context, *Value) (*Response, error)
    Ping(context.Context, *Heartbeat) (*Heartbeat, error)
}
```

Listing 1: The `SinglePaxosServer` interface that Paxos replicas must implement.

Our goal in this paper is to provide a framework for generating test cases to validate the correctness of the Gorums implementation itself, in addition to different quorum function and quorum call implementations for our Gorums-based Paxos implementation.

4.2. Implementing single-decree Paxos using Gorums

We have implemented the single-decree Paxos protocol as our system under test, using Gorums and the Go programming language. The system consists of $n = 2f + 1$ replicas that run the Paxos protocol, taking client requests as input, aimed at reaching consensus on a single output response. The implementation corresponds to the CPN model discussed in §2 and §3.

In our implementation, we first define a set of RPC service methods for Paxos using the interface description language (IDL) of protocol buffers [30]. This IDL is then supplied to the Gorums code generator, which generates the code necessary to invoke quorum calls for the methods defined in the IDL. Each of the Paxos replicas must implement the `SinglePaxosServer` interface shown in Listing 1, which is generated from the IDL. The methods Prepare(), Accept() and Commit() in this interface represent Paxos quorum calls that can be invoked by the different replicas in order to access and update each other's Paxos state.

In addition to the Paxos methods mentioned above, the `SinglePaxosServer` interface also contains ClientHandle() and Ping(). The former is a quorum call used by clients to communicate their proposed value to the Paxos replicas and receive the decided value. Recall that multiple clients can propose a value, possibly simultaneously, but only one of the proposed values will be decided, and returned to all clients. The Ping() is simply a regular RPC call used by the failure detector to determine if a replica has failed.

In the following, we explain the main control flow of the single-decree Paxos protocol, as shown in Listing 2; ignoring error handling and `ctx` initialization. On Line 3 of the `Proposer`, the Prepare() quorum call sends a ⟨PREPARE⟩ message to the `Acceptors`, whom return ⟨PROMISE⟩ messages back to the `Proposer`. Once a quorum of promises has been received,

```
1  func (p *Proposer) runPaxosPhases() error {
2      preMsg := &PrepareMsg{Rnd: crnd}
3      prmMsg, err := p.config.Prepare(ctx, preMsg)
4      if prmMsg.GetVrnd() != Ignore {
5          p.cval = prmMsg.GetVval()
6      }
7      accMsg := &AcceptMsg{Rnd: crnd, Val: p.cval}
8      lrnMsg, err := p.config.Accept(ctx, accMsg)
9      ackMsg, err := p.config.Commit(ctx, lrnMsg)
10     return nil
11 }
```

Listing 2: `Proposer`'s code for Paxos phases, slightly simplified, and without error handling.

```
type QuorumSpec interface {
    PrepareQF(replies []*PromiseMsg) (*PromiseMsg, bool)
    AcceptQF(replies []*LearnMsg) (*LearnMsg, bool)
    CommitQF(replies []*Empty) (*Empty, bool)
    ClientHandleQF(replies []*Response) (*Response, bool)
}
```

Listing 3: The QuorumSpec interface must be implemented to process replies.

```
1  type PaxosQSpec struct {
2      quorum int
3  }
4
5  func (q PaxosQSpec) PrepareQF(replies []*PromiseMsg) (*PromiseMsg,bool) {
6      if len(replies) < q.quorum {
7          return nil, false
8      }
9      reply := &PromiseMsg{Rnd: replies[0].GetRnd()}
10     for _, r := range replies {
11         if r.GetVrnd() >= reply.GetVrnd() {
12             reply.Vrnd = r.GetVrnd()
13             reply.Vval = r.GetVval()
14         }
15     }
16     return reply, true
17 }
```

Listing 4: The `PrepareQF` processes ⟨Promise⟩ replies from replicas.

the Prepare() quorum call returns with a single combined ⟨Promise⟩ message. We explain later in this section, how we leverage Gorums's quorum function abstraction to determine whether or not a quorum has been received, and how to combine the replies into a single ⟨Promise⟩ message.

Next, the `Proposer` determines from the ⟨Promise⟩ if any of the `Acceptors` have voted in a previous round, *vrnd*. If so, the corresponding value from the ⟨Promise⟩ message (Line 5) that was voted for, must also be used by the `Proposer` when constructing its ⟨Accept⟩ message on Line 7. Otherwise, the `Proposer` uses the value *cval* that it received from a client.

At this stage the `Proposer` invokes the Accept() quorum call, asking the `Acceptors` to choose the value included in the ⟨Accept⟩ message. The `Acceptors` respond back with a ⟨Learn⟩ message, followed by the `Proposer` invoking the Commit() quorum call to propagate the decision to the `Learners`, which concludes the protocol.

We have implemented the `SinglePaxosServer` interface methods on an object of type `PaxosReplica`, encapsulating the state and behavior of the Paxos agents: `Proposer`, `Acceptor`, and `Learner`. The behavior of each agent corresponds to different CPN models in §3. Further, the `PaxosReplica` takes care of dispatching the method calls to their respective Paxos agents.

We now turn our attention to the handling of replies from quorum calls. For each quorum call defined in the IDL, Gorums adds a quorum function signature to an interface called `QuorumSpec`, as shown in Listing 3. This interface must be implemented by the protocol developer; Listing 4 shows the implementation of the `PrepareQF` quorum function. These methods are implemented on the `PaxosQSpec` type, which holds information about the quorum size (Line 2).

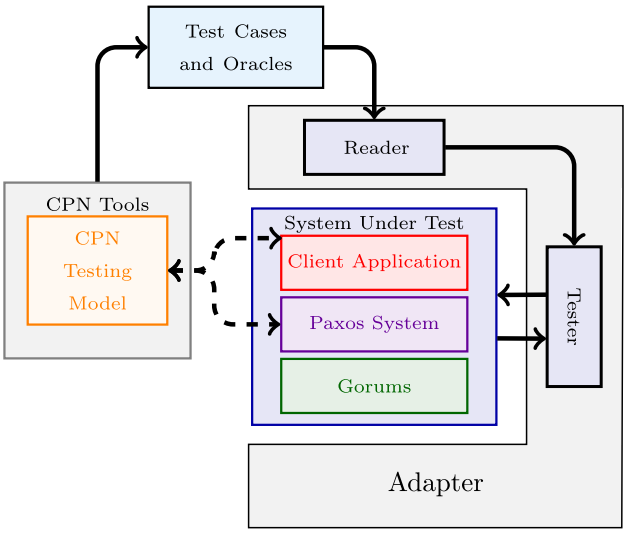


Fig. 15. Overview of the test framework.

`PrepareQF` is called by the Gorums runtime with the set of replies that have been received so far; it is called once for each reply. In the first part (Lines 6–8), we check if there are enough replies to return from the quorum call, or return `false` to signal to Gorums that we must wait for more replies.

If enough replies have been received, we continue to construct a combined ⟨Promise⟩ message by examining all the replies, and picking the value, *vval*, from the ⟨Promise⟩ message with the highest voted round, *vrnd*. If such a locked-in value is found in the replies, this means that the `Proposer` is constrained and must continue to use this value in the remainder of the protocol. Otherwise, the `Proposer` is unconstrained, and can pick its own client value.

Similar constructs are used for all the methods in the QuorumSpec interface, but we do not show them here. However, we note that one of the benefits of using Gorums's quorum functions is that they are amenable to unit testing.

5. Test case execution

To perform model-based testing of our Paxos implementation described in §4.2, we have implemented a client application, which together with the Paxos implementation and Gorums constitute the SUT. Fig. 15 gives an overview of our test framework, which consists of CPN Tools and a test adapter. Our test approach involves three steps: (a) use CPN Tools to construct a test model of our SUT; (b) perform simulation-based test case generation by using the MBT/CPN library to generate test cases with oracles represented in an XML format; (c) implement a test adapter to execute the generated test cases on the SUT, and compare the test results against generated oracles.

5.1. The test adapter

A central part of our test approach is the development of a test adapter which can execute the system and unit test cases generated from CPN Tools using our MBT/CPN library [10] (discussed in §6). The test adapter consists of a reader and a tester, both implemented in Go. The reader of the adapter can read test cases with oracles in the XML format generated from the CPN test model. The tester component has been implemented using the *testing* package from the Go standard library. Go's testing infrastructure comprises the `go test` command which allows us to simply run and execute our generated tests and obtain pass/fail information for each test case execution. Moreover, the Go testing infrastructure includes a tool which can be used to evaluate our approach by measuring the statement coverage for both unit and system tests.

5.2. Test case execution

We distinguish between unit and system tests for our SUT. The unit tests are used to test the central protocol logic used to implement the single-decree Paxos protocol, such as quorum functions discussed in §4. The system tests are used to test the complete Paxos implementation and Gorums library with clients. This separation provides a modular approach to testing. Additionally, under system tests, we consider failure scenarios for the Paxos replicas when in different Paxos phases, cf. Fig. 8.

5.2.1. Unit tests

The test adapter implements a Go-based tester that can execute the unit tests obtained from the reader. The tester invokes the methods to be tested with the supplied input values, and upon completion compares the results against the

```
<Test Name="TestPrepareQF">
  <TestCase ID="1">
    <TestValues>
      <PromiseMsg>
        <Rnd>0</Rnd>
        <Vrnd>0</Vrnd>
        <Vval></Vval>
      </PromiseMsg>
      <PromiseMsg>
        <Rnd>0</Rnd>
        <Vrnd>0</Vrnd>
        <Vval></Vval>
      </PromiseMsg>
    </TestValues>
    <TestOracles>
      <Quorum>true</Quorum>
      <PromiseMsg>
        <Rnd>0</Rnd>
        <Vrnd>0</Vrnd>
        <Vval></Vval>
      </PromiseMsg>
    </TestOracles>
  </TestCase>
</Test>
```

Fig. 16. XML format for PrepareQF().

test oracle's expected output, also obtained from the reader. The unit tests can be performed without running the Paxos protocol and clients. The methods we consider for unit tests include PrepareQF() and AcceptQF(), discussed in §3 and §4.2. Fig. 16 shows an excerpt from the XML representation of a test case for PrepareQF(), which corresponds to a test case where Paxos is configured with three replicas and the quorum size is two. The test input for the PrepareQF() method in the test case is two ⟨PROMISE⟩ messages with values for the fields *Rnd*, *Vrnd* and *Vval*. The expected output of the PrepareQF() is a ⟨PROMISE⟩ message together with the Quorum boolean *true*, indicating that a quorum was obtained for these input messages.

5.2.2. System tests

Execution of the system tests requires that the Paxos replicas are running and ready to handle the requests from clients so that we can test the complete system including the Gorums library.

Therefore, for system tests, the tester first starts the Paxos replicas and then iterates through the test cases obtained from the reader. For each test case, the tester starts clients in order to send client requests to the Paxos replicas. Each client has a single request value to send for consensus. As an example, the test adapter can execute two clients concurrently to send their requests to the Paxos replicas. After the Paxos replicas have decided, a response value is sent back to the clients. The tester checks whether the response for each client belongs to the expected responses (oracles) and whether the responses are the same for all clients, i.e., the consensus was reached.

In addition to testing success scenarios, we also test scenarios with different types of failures. This includes forcing the failure detector to timeout, triggering a new leader to be promoted. In this way, we can test leader changes and fault tolerance of the Paxos protocol. To make the implementation amenable for such failure scenarios during system test execution, our test adapter must be able to observe the messages exchanged between the replicas, and to interfere, for example, in a test case where we simulate a lost message or trigger a timeout.

We have considered three major harnessing approaches below for how we can effectively test a particular scenario, and we motivate our final choice. Further details on the first two approaches and their pros and cons can be found in [31].

In the first approach, one would isolate the involved (Unix) processes in individual, networked, containers or virtual machines, and if necessary interfere with the environment by, e.g., introducing network partitions. This is a heavy-weight approach, where a lot of implementation effort will have to be spent on manipulating the environment based on a test case description. Also, the test case adapter coordinating the environment needs an understanding of the messages to be exchanged between replicas, so that it can decide that a particular setup has now been reached and it should interfere.

The second approach is more light-weight in that the Paxos replicas would connect to the test adapter instead of directly to each other. The test adapter can then observe the protocol and either relay message, or introduce faults [32]. This approach can reuse marshaling logic in the test adapter, which makes analyzing the message content easier than in the virtual machine approach above.

Our approach is even more light-weight in that we do not use an external test adapter. Instead, to track the state of a replica, we compile an instrumented version of the server that contains several points for test case interaction. By using this approach, we can use test cases to describe not only the successful scenarios, but also different failure scenarios and guide the test case execution. As an example, consider a Paxos configuration with three replicas. A test case may contain events that cause the leader to fail during either the first or the second phase of the Paxos protocol. After such a failure, a new leader will eventually emerge, restarting the Paxos phases. In a configuration with five replicas, a test case can, e.g., be

```
<Test Name="systemtest">
  <TestCase ID="1">
    <TestValues>
      <ClientPropose>M1</ClientPropose>
      <ClientPropose>M2</ClientPropose>
      <P1Failure>1</P1Failure>
    </TestValues>
    <TestOracles>
      <Leader>0</Leader>
      <Leader>1</Leader>
      <Response>M1</Response>
      <Response>M2</Response>
    </TestOracles>
  </TestCase>
</Test>
```

Fig. 17. XML format for testing a three-way replicated Paxos system.

configured to let the first leader fail in the first phase, and after the second leader emerges and the Paxos phases restart, the second leader can be made to fail in the second phase. Finally, the third leader can restart and complete the Paxos phases successfully.

To enable the test adapter to know when it is possible to inject a failure, we have instrumented the `Proposer` with an `AdapterConnector` to communicate the `Proposer`'s state, such as the current leader and which Paxos phases have completed, to the test adapter. Moreover, between each state change, the `Proposer` will wait for a decision from the test adapter to determine if the current Paxos phase should fail, e.g. triggering a leader failure. The decisions made by the test adapter regarding failures of the Paxos phases are configured for each test case in the XML file. Fig. 17 shows an example of a test case for the Paxos protocol with three replicas, where there is a failure in the first Paxos phase. The test input for this example consists of two clients sending requests concurrently to the Paxos replicas. The test oracles include the legal responses from Paxos replicas, and the expected leaders. Leader 0 is the first leader, and after it fails, leader 1 becomes the new leader. The test adapter checks whether the correct leaders are chosen, and whether the response returned to each client belongs to the set of legal responses. Furthermore, it also tests whether the responses obtained by all clients are equal, so that we can determine if they have reached consensus.

6. Model validation and test case generation

For the test case generation we rely on the MBT/CPN library [10], which we have developed as an extension to CPN Tools. The MBT/CPN library is based on extracting test cases from execution sequences of the CPN model by partially observing occurring events. MBT/CPN supports both state space and simulation-based test case generation. State space-based test case generation works for finite-state models and is based on computing all reachable state and state changes of the CPN model. Simulation-based test case generation is based on running a set of simulations and extracting test cases from the corresponding set of executions.

The CPN test model for the Paxos protocol has an infinite state space and also for restricted and representative configurations with a finite state space, state-based test case generation is infeasible due to the state explosion problem. We therefore only use state space for validating the CPN model for small configurations (see §6.1) in order to gain confidence in the correctness of the test generation CPN model. For the test case generation itself, we rely on simulation-based test case generation due to the high complexity of the Paxos protocol.

6.1. Model validation

A distinct advantage of relying on formal models such as CPNs for test case generation is that restricted configurations of the test case generation model with a finite state space can be verified using model checking prior to test case generation. This can be used to increase confidence in the correctness of the test case generation model and the generated test cases. To obtain configurations of the Paxos CPN test generation model with a finite state space, we have bounded the behavior of the Paxos agent roles such that only a finite number of messages can be generated in the system.

Specifically, we consider configurations of our CPN model with two clients, where each client can send one client request message (modeled as a string) into the Paxos system. These two request messages can be sent in any order, and the Paxos system then makes a decision on which client request message should be chosen and handled. The model terminates when both clients have received the decision response from the Paxos system. For the Paxos agent roles, we have limited the number of messages when executing the Paxos phases by configuring an upper bound of one on the number of timeouts/failures. This is done by means of place FDControl discussed in §3.1 and shown in Fig. 8. The most complex scenario currently covered is where the first Paxos phase fails once, then the Paxos system restarts the first phase; but this time the second phase fails once and the Paxos system restarts again, and then Paxos completes successfully for both phases. This scenario involves the `Proposer` (leader) sending the ⟨Prepare⟩ message three times, the ⟨Accept⟩ message two times, and the ⟨Commit⟩ message once.

In other words, we explicitly model failure scenarios where messages timeout or get lost in particular phases of the protocol, and combinations thereof. After the associated restarts of the Paxos protocol in the presence of these failures, the model lets the run complete successfully without further errors.

We have used the model checker and ASK-CTL library available in CPN Tools to verify that the CPN model (in the restricted configurations) satisfies the correctness properties S1–S3 and L1–L2 as formulated in §2. The ASK-CTL library makes it possible to specify temporal properties in a state and event-oriented variant of the computation tree logic (CTL). Below we show how the behavioral properties can be specified in CTL relative to the developed test case generation CPN model. We use $M(p)$ to denote the marking (multi-set of tokens) on a place p in the marking (state) M. For a token (value) t, we use $t \in M(p)$ to denote that t is a token on place p in the marking M.

S1 Only a proposed value may be chosen: To check this property we consider the place ServerResponse in Fig. 2. Proposed values are represented as tokens on place ClientRequest in the initial marking (state), and the chosen consensus value will appear as a token on place ServerResponse. The property can therefore be formulated in CTL as:

$$\mathbf{AG}\,(t \in M(\mathsf{ServerResponse}) \Rightarrow t \in M_0(\mathsf{ClientRequest}))$$

S2 Only a single value is chosen: As any chosen value will appear as a token on place ServerResponse we can verify this property by checking that there is at most one token on this place in any reachable state. In CTL this can be formulated as:

$$\mathbf{AG}\,(|M(\mathsf{ServerResponse})| \leq 1)$$

S3 Only a chosen value may be learned by a correct replica: We consider the tokens on AcceptorState (Fig. 9) of the form $(r, rnd, vrnd, v)$ where the first component specifies the replica and the last component specifies the chosen value. The value learned by each replica will appear as tokens on place Response (Fig. 3), where the first component specifies the replica and the second component specifies the learned value. To account only for correct replicas, we consider the fusion place FailedReplica (Fig. 8) and restrict the property to replicas not present on this place. The property can therefore be checked using the following CTL formula where R denotes the set of replicas:

$$\mathbf{AG}(\forall r \in R \setminus M(\mathsf{FailedReplica}) :$$
$$(r, v) \in M(\mathsf{Response}) \Rightarrow \exists(r', rnd, vrnd, v) \in M(\mathsf{AcceptorState}))$$

L1 Some proposed value is eventually chosen: For this property we can check that eventually a token will be put on place ServerResponse. In CTL this can be formulated as:

$$\mathbf{AG}\,\mathbf{AF}\,(M(\mathsf{ServerResponse}) \neq \emptyset)$$

L2 Once a value is chosen, correct replicas eventually learn it: We consider the place AcceptorState holding any chosen value, and check that this value is eventually learned by non-failing replicas by considering the place Response in Fig. 3. In CTL this property can be formulated as:

$$\mathbf{AG}\,(\exists(r, rnd, vrnd, v) \in M(\mathsf{AcceptorState}) \Rightarrow$$
$$\mathbf{AF}(\forall r \in R \setminus M(\mathsf{FailedReplica}) : (r, v) \in M(\mathsf{Response}))$$

We have executed the above queries against the test case generation CPN model configured with two replicas which yields a relatively small state space with less than 2000 states. In the process of checking these properties we found a number of minor modeling errors that we were then able to correct. In particular, we use the support in CPN Tools to obtain error traces (in case a property was violated) which helped in identifying the source of the problem. Even if the Paxos model is too complex to conduct model checking for larger configurations (due to the state space size), being able to verify the model for smaller configurations increases the confidence in the correctness of our test case generation for larger configurations of the Paxos protocol.

6.2. Test case specification

Test case generation from the CPN model requires identification of *observable events* originating from occurrences of transitions. A test case is comprised of observable events, where the input events represent stimuli to the system and the output events represent the expected outputs used as test oracles to determine the pass/failure of a test case. The formal foundation used to check whether the execution of the SUT conforms to the specification as provided by the test case is hence based on trace equivalence.

The generation of test cases with MBT/CPN requires an implementation of a *test case specification* defined by the Standard ML signature (interface) shown in Listing 5.

```
signature TCSPEC = sig
  val detection   : Bind.Elem -> bool;
  val observation : Bind.Elem -> TCEvent list;
  val format      : TCEvent    -> string
end;
```

Listing 5: Signature for test case specification.

```
fun detection (Bind.DecidedValue'Request _) = true
  | detection (Bind.DecidedValue'Apply_RequestQF _) = true
  | detection (Bind.PhaseOne'Process_PromiseMsg _) = true
  | detection (Bind.PhaseTwo'Process_LearnMsg _) = true
  | detection _ = false;

exception obsExn;
fun observation (Bind.DecidedValue'Request (_,b)) = [InEvent (SYS_Propose (#value b))]
  | observation (Bind.DecidedValue'Apply_RequestQF (_,b) = [OutEvent (SYS_Decide (#value b))]
  | observation (Bind.PhaseOne'Process_PromiseMsg (_,b) = [InEvent (SYS_P1Failure (#crnd b))]
  | observation (Bind.PhaseTwo'Process_LearnMsg (_,b) = [InEvent (SYS_P2Failure (#rnd b)))]
  | observation _ = raise obsExn;
```

Listing 6: Implementation of test case specification for system level tests.

The type `Bind.Elem` is an existing data type in CPN Tools representing binding elements, i.e., a transition and an assignment of values to the variables of the transition. The type `TCEvent` is the type defined for observable events. The *detection function* is a predicate on binding elements that evaluates to true for binding elements representing observable events. The purpose of the *observation function* is to map an observable binding element into an observable input or output event belonging to the `TCEvent` type. The observation function may return a list of observable events in case one might want to split a binding element into several observable events in the test case. Finally, the *formatting function* maps observable events into a string representation which is used in order to export the test cases into files.

For the Paxos protocol we generated both system test and unit tests. The system level test is concerned with the proposed values, chosen value, selected leaders, and failure of replicas. The unit test are concerned with testing the quorum functions, which forms the core of the Gorums-based implementation. Listing 6 shows a slightly simplified implementation of the detection and observation function for system level tests. We omit the formatting function as the XML format for test cases is already described in §5.2.

The first two binding elements for which the detection function returns true correspond to events representing the proposal and choice of a value. The two next binding elements correspond to events representing replica failures. The observation function then generates the observable events, which can be either an `InEvent` representing stimuli to the system or an `OutEvent` representing expected outputs. The implementation of the test case specification for unit tests covers the prepare, accept, and commit quorum functions and the implementation is similar to the system test case specification.

6.3. Experimental results on statement coverage

We have used statement coverage to evaluate the quality of our test case generation. Several other metrics exist to assess test coverage, but currently only statement coverage is supported by the Go tool chain. Table 1 summarizes the experimental results obtained using simulation-based test case generation for the Paxos protocol. We have considered Paxos configurations with 3 and 5 replicas and generated 1, 2, 5 and 10 simulation runs of the CPN model. As we did not see any increase in the number of test cases by going from 5 to 10 simulations, we did not increase the number of simulation runs further. The table shows the coverage obtained for the different subsystems of our Paxos implementation. Note that the Unit tests are for the quorum functions and hence not applicable for the other subsystems. The two numbers written below System tests and Unit tests gives the total number of test cases generated for 3 and 5 replica configurations, respectively. The test case generation for each configuration considered took less than 10 seconds, and the execution of each test case took less than one minute.

The results show that, for the configuration with both 3 and 5 replicas, the statement coverage of unit tests for Prepare and Accept quorum functions are up to 90% and 85.7%, respectively. For the system tests, the statement coverage for Prepare, Accept and Commit quorum calls reaches 83.9%, respectively; the results of statement coverage for Prepare and Accept quorum functions are up to 100%; for the Paxos implementation (Paxos core in the table), the Proposer module's statement coverage reaches 97.4%; the statement coverage of the Acceptor module is up to 100%; the statement coverages of the Failure Detector and Leader Detector modules reach 75.0% and 91.4%, respectively; the statement coverage of the Paxos replica module (discussed in §4.2) reaches 91.4%; for the Gorums library as a whole, the highest statement coverage reaches 51.8%. The results and test cases considered above validate that the implementation of the single-decree Paxos system and the Gorums framework work in both correct scenarios and scenarios involving failures of replicas. The reason for the lower coverage results of the Gorums library is that Gorums contains code generated by Gorums's code generator, and among them, various auxiliary functions and error handling code that are not used by our current implementation. The total number of lines of code for the SUT is approximately 3890 lines, which include generated code by Gorums's code

Table 1
Experimental results for test case generation and execution.

Subsystem	Component	System tests	Unit tests
		Test cases for 3/5 replicas	
		15/38	74/424
		Coverage	
Gorums library		51.8%	–
Paxos core	Proposer	97.4%	–
	Acceptor	100.0%	–
	Failure Detector	75.0%	–
	Leader Detector	91.4%	–
	Replica	91.4%	–
Quorum calls	Prepare	83.9%	–
	Accept	83.9%	–
	Commit	83.9%	–
Quorum functions	Prepare	100.0%	90.0%
	Accept	100.0%	85.7%

generator (around 3150 lines), the code for Paxos replica (around 110 lines), the client code (around 80 lines), the Proposer code (around 170 lines), the Acceptor code (around 40 lines), the code for failure detector (around 170 lines), the code for leader detector (around 100 lines), and the code for quorum functions (around 70 lines).

As part of analyzing the test results and executing generated test cases, we have discovered bugs in the implementation of the Paxos protocol, which are not captured by using manually written table-driven tests in Go. We have found bugs related to: the leader detector elects a wrong leader; only the leader's failure detector is executed; the elected new leader obtains a wrong round number; clients cannot receive responses from the Paxos replicas; the Paxos system can only handle one request from one client; and after the current leader fails, the failed leader executes the Paxos phases again. This shows how our MBT approach is able to detect non-trivial programming errors in complex distributed systems protocols.

7. Related work

Chubby [33] was one of the first implementations of Paxos that were deployed in a production environment, and thus were extensively tested. The authors highlight that at the time (2007), it was unrealistic to prove correct a real system of that size. Thus to achieve robustness, they adopted meticulous software engineering practices, and tested their system thoroughly. One of their testing strategies was to test their implementation when subjected to a random sequence of network outages, message delays, timeouts, process crashes and recoveries, schedule interleavings, and so on. Using our CPN model and our generated tests, we aim to test many of the same attributes in a more systematic manner.

Modbat is an MBT tool implemented in Scala and hence compatible with Java bytecode-based applications [34]. Models are specified as annotated, non-deterministic extended finite state machines. Modbat explores the transition system and executes the calls specified on the transitions. It has been used successfully in a similar setting as ours on the ZooKeeper distributed coordination service. It explores different possible interleavings and non-deterministic outcomes due to scheduling decisions or network communication in the real system which are judged by an oracle essentially implementing a model checking component. Unlike our CPN models, the specifications are not for consumption by other tools such as model checkers, nor is there an interactive component that allows exploring a particular execution of the model. As in our approach, it requires some manual effort connecting the engine to the SUT.

A testing approach for true concurrency using I/O Petri nets has been discussed by Ponce de León et al. [35]. The authors define a concurrent conformance relation for input–output labeled transitions systems, IOLTS. They present a test case selection algorithm using criteria such as all paths of length n, or traversing each basic behavior a certain number of times. Since test case selection is also a challenge in our setting, it remains an open question how their unfoldings would work in our CPN setting.

MBT has been used with success (as measured through productivity gain) in Microsoft's Protocol Documentation Quality Assurance Process. Grieskamp et al. [31] used Spec Explorer on protocols, where a so-called model program describes the test case, including how to check an observation against a possibly non-deterministic outcome. The main difference to our work is that their model programs are rule-based, and as such only get a visual representation as a graph through state space exploration. Our CPN models give developers a better overview as they directly link client- and server interactions. Spec Explorer uses the coordination language Cord for slicing models into tractable subsets of test cases that may impact coverage and completeness, but not correctness. We have not yet tackled the issue of test case selection, relying on user interaction through simulation when state space exploration becomes infeasible.

A CPN-based test generation approach is proposed by Liu et al. [6]. This approach requires defining a conformance testing-oriented CPN (CT-CPN) model and a PN-ioco relation which specifies how an implementation conforms to its specifications. Furthermore, this approach uses simulation-based test case generation algorithm for the CT-CPN model. In our

approach, on the other hand, test cases can be generated directly by using a simulation-based approaches for an existing implementation of the SUT. In addition, Wu, Schnieder, and Krause [7] use a model-based test generation technique based on CPNs to verify a module of a satellite-based train control system. They use CPN Tools to generate the reachability graph of the test model and then use state space analysis with CPN Tools to extract the expected output of each test case from the path of the graph. However, their approach does not support simulation-based test case generation, which is of essential for scalability. Zheng et al. [8] provide a technique for test cases and sequences generation. In their method, two algorithms are used to generate test cases and sequences from a CPN model of the SUT. The CPN model is first used as input to their APCO algorithm to generate an initial set of test cases. These test cases can then be converted to test sequences by using their algorithm. After that, the set of original test cases and test sequences can be exported as XML formatted files. They have applied their technique to a radio module in a centralized railway control system. In contrast to our approach, Zheng et al. do not consider testing any failure scenarios of the system, do not handle any concurrent execution of the system, and their approach has not been used to validate any distributed systems.

Formal verification of protocols for distributed systems tackles protocols on a more abstract level, and is interested in finding flaws and inconsistencies primarily in the specification. Such approaches are not necessarily interested in a correct implementation, and only rarely can executable code directly or automatically be derived from the specification. Formal verification of such complex systems often suffers from undecidability issues that require careful management of any automation (see [36]), or substantial effort to encode the system in a decidable fragment (see Padon et al. [37] for their encoding of Paxos and Multi-Paxos in EPR, the effectively-propositional fragment of first-order logic). We see our approach of testing a concrete implementation as orthogonal to approaches that aim to validate the correctness of a protocol in general: frequently, the final, often manual, step of actually programming a proven-as-correct algorithm introduces mistakes, and also generated code may suffer from problems or assumptions about the underlying infrastructure (see e.g. Fonseca's analysis of IronFleet among others [38]).

8. Conclusions and future work

The main contribution of our work is an MBT approach for advanced distributed systems protocols based on formal modeling. As we have illustrated on the Paxos protocol, application of our approach includes constructing a CPN testing model for the system under test, executing simulation-based test case generation algorithms, and applying a test case execution framework which combines test cases obtained from CPN Tools and a test adapter. Our experiments with this testing approach on a single-decree Paxos protocol implemented by the Gorums framework have demonstrated good code coverage and considered both unit and system tests. Furthermore, for the system tests, we have considered not only tests representing successful, non-faulty executions of the Paxos protocol, but also tests in which replicas may fail during the protocol's execution, and show that the implementation can handle these failure scenarios. We have shown that our approach detected errors and bugs in the Paxos implementation.

An attribute of our testing approach is that the constructed CPN testing model can also help us to obtain a better understanding of a complex protocol to be implemented. Furthermore, our Paxos CPN testing model can also serve as a basis for MBT of multi-decree Paxos and other fault-tolerant distributed systems implemented with the abstractions of the Gorums framework. For example, given a distributed system implemented by the Gorums framework, it is only the implementation of the quorum functions that needs to be changed when modeling the behaviors of quorum calls and quorum functions.

Another attribute is that we have used simulation-based test case generation for the Paxos system with differently sized configurations, e.g. with three or five replicas. Another contribution worth mentioning is our implementation of single-decree Paxos using Gorums. It is well-known that the Paxos protocol is difficult to understand and implement correctly. However, by leveraging the Gorums framework and its abstractions, our single-decree Paxos implementation is simpler and hence more reliable than it would be without Gorums. Additionally, we expect that it will be relatively easy to extend our implementation to multi-decree Paxos.

Our work opens up several paths for future work. For MBT of both successful scenarios of our Paxos protocol and scenarios involving failure injections of replicas, we have obtained good statement coverage results for unit and system tests. However, we need to consider more of Gorums's code paths so that we can increase the results of the coverage for the Gorums library itself. In order to do this, we need to test the Paxos protocol under additional failures scenarios and adverse conditions, such as network errors and partitions. This will require extensions to the current CPN testing model and XML format to describe and configure such failure scenarios so that we can use the generated test cases to guide the test case execution based on different failures scenarios. This also requires an extension to the test adapter such that it can execute the Paxos system under test with additional configurations in test cases to handle the failure scenarios. In addition to the single-decree Paxos, we also plan to evaluate our testing approach on additional complex protocols in order to evaluate the generality of our testing approach. In the short term, we are extending our current CPN testing model and Go implementation to a multi-decree Paxos protocol, and then perform MBT for such a complex Paxos system.

References

[1] Jepsen, Distributed systems safety analysis, http://jepsen.io.

[2] M. Utting, B. Legeard, Practical Model-Based Testing: A Tools Approach, Elsevier, 2010.
[3] K. Jensen, L.M. Kristensen, Coloured Petri Nets: a graphical language for modelling and validation of concurrent systems, Commun. ACM 58 (6) (2015) 61–70.
[4] CPN Tools, CPN Tools homepage, http://www.cpntools.org, 2017.
[5] L.M. Kristensen, K.I.F. Simonsen, Applications of Coloured Petri Nets for Functional Validation of Protocol Designs, Springer, 2013, pp. 56–115.
[6] J. Liu, X. Ye, J. Li, Colored Petri Nets model based conformance test generation, in: IEEE Symp. on Computers and Communications, ISCC, IEEE, 2011, pp. 967–970.
[7] D. Wu, E. Schnieder, J. Krause, Model-based test generation techniques verifying the on-board module of a satellite-based train control system model, in: 2013 IEEE Intl. Conf. on Intelligent Rail Transportation Proceedings, 2013, pp. 274–279.
[8] W. Zheng, C. Liang, R. Wang, W. Kong, Automated test approach based on all paths covered optimal algorithm and sequence priority selected algorithm, IEEE Trans. Intell. Transp. Syst. 15 (6) (2014) 2551–2560, https://doi.org/10.1109/TITS.2014.2320552.
[9] L.M. Kristensen, V. Veiset, Transforming CPN models into code for TinyOS: a case study of the RPL protocol, in: Proc. of ICATPN'16, in: Lecture Notes in Computer Science, vol. 9698, Springer, 2016, pp. 135–154.
[10] MBT/CPN repository, https://github.com/selabhvl/mbtcpn, Aug 2018.
[11] L. Lamport, The part-time parliament, ACM Trans. Comput. Syst. 16 (2) (1998) 133–169.
[12] L. Lamport, Fast Paxos, Distrib. Comput. 19 (2) (2006) 79–103, https://doi.org/10.1007/s00446-006-0005-x.
[13] L. Lamport, D. Malkhi, L. Zhou, Vertical Paxos and primary-backup replication, in: Proceedings of the 28th ACM Symp. on Principles of Distributed Computing, PODC '09, ACM, Calgary, AB, Canada, 2009, pp. 312–313.
[14] I. Moraru, D.G. Andersen, M. Kaminsky, There is more consensus in egalitarian parliaments, in: ACM SIGOPS 24th Symp. on Operating Systems Principles, SOSP '13, 2013.
[15] H. Meling, K. Marzullo, A. Mei, When you don't trust clients: Byzantine proposer fast Paxos, in: 32nd IEEE International Conference on Distributed Computing Systems, ICDCS, IEEE, 2012, pp. 193–202.
[16] M. Burrows, The Chubby lock service for loosely-coupled distributed systems, in: Proc. of the 7th Symp. on Operating Systems Design and Implementation, OSDI '06, USENIX Association, 2006, pp. 335–350.
[17] D.F. Bacon, N. Bales, N. Bruno, B.F. Cooper, A. Dickinson, A. Fikes, C. Fraser, A. Gubarev, M. Joshi, E. Kogan, A. Lloyd, S. Melnik, R. Rao, D. Shue, C. Taylor, M. van der Holst, D. Woodford, Spanner: becoming a SQL system, in: Proc. of the 2017 ACM Intl. Conf. on Management of Data, SIGMOD '17, ACM, Chicago, Illinois, USA, 2017, pp. 331–343.
[18] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, M. Deardeuff, How Amazon web services uses formal methods, Commun. ACM 58 (4) (2015) 66–73, https://doi.org/10.1145/2699417.
[19] H. Meling, L. Jehl, Tutorial summary: Paxos explained from scratch, in: R. Baldoni, N. Nisse, M. van Steen (Eds.), 17th International Conference on Principles of Distributed Systems, OPODIS, in: Lecture Notes in Computer Science, vol. 8304, Springer, 2013, pp. 1–10.
[20] T.E. Lea, L. Jehl, H. Meling, Towards new abstractions for implementing quorum-based systems, in: Proc. of 37th IEEE Intl. Conf. on Distributed Computing Systems, ICDCS, 2017, pp. 2380–2385.
[21] CPN testing model of the single-decree Paxos, http://dkan.isp.uni-luebeck.de/story/automated-tcs-gen-cpns, March 2018.
[22] L. Lamport, Paxos made simple, ACM SIGACT News 32 (4) (2001) 18–25.
[23] J.-P. Martin, L. Alvisi, Fast Byzantine consensus, IEEE Trans. Dependable Secure Comput. 3 (3) (2006) 202–215.
[24] M.J. Fischer, N.A. Lynch, M.S. Paterson, Impossibility of distributed consensus with one faulty process, J. ACM 32 (2) (1985) 374–382, https://doi.org/10.1145/3149.214121.
[25] T.D. Chandra, V. Hadzilacos, S. Toueg, The weakest failure detector for solving consensus, J. ACM 43 (1996) 685–722, https://doi.org/10.1145/234533.234549.
[26] H. Attiya, A. Bar-Noy, D. Dolev, Sharing memory robustly in message-passing systems, J. ACM 42 (1) (1995) 124–142.
[27] L. Jehl, R. Vitenberg, H. Meling, SmartMerge: a new approach to reconfiguration for atomic storage, in: Y. Moses (Ed.), Distributed Computing – 29th Intl. Symp., DISC 2015, in: Lecture Notes in Computer Science, vol. 9363, Springer, 2015, pp. 154–169.
[28] M. Vukolić, Quorum Systems: With Applications to Storage and Consensus, Synthesis Lectures on Distributed Computing Theory, vol. 3 (1), Morgan & Claypool Publishers, 2012.
[29] Google Inc., gRPC remote procedure calls, http://www.grpc.io.
[30] Google Inc., Protocol buffers, http://developers.google.com/protocol-buffers.
[31] W. Grieskamp, N. Kicillof, K. Stobie, V. Braberman, Model-based quality assurance of protocol documentation: tools and methodology, Softw. Test. Verif. Reliab. 21 (1) (2011) 55–71, https://doi.org/10.1002/stvr.427.
[32] H. Meling, A framework for experimental validation and performance evaluation in fault tolerant distributed system, in: Workshop on Dependable Parallel, Distributed and Network-Centric Systems, DPDNS, IEEE, 2007, pp. 1–8.
[33] T.D. Chandra, R. Griesemer, J. Redstone, Paxos made live: an engineering perspective, in: Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC '07, ACM, 2007, pp. 398–407.
[34] C. Artho, Q. Gros, G. Rousset, K. Banzai, L. Ma, T. Kitamura, M. Hagiya, Y. Tanabe, M. Yamamoto, Model-based API testing of apache ZooKeeper, in: 2017 IEEE Intl. Conf. on Software Testing, Verification and Validation, ICST, 2017, pp. 288–298.
[35] H. Ponce de León, S. Haar, D. Longuet, Model-based testing for concurrent systems: unfolding-based test selection, Int. J. Softw. Tools Technol. Transf. 18 (3) (2016) 305–318, https://doi.org/10.1007/s10009-014-0353-y.
[36] C. Hawblitzel, J. Howell, M. Kapritsos, J. Lorch, B. Parno, M.L. Roberts, S. Setty, B. Zill, IronFleet: proving practical distributed systems correct, in: Proceedings of the ACM Symposium on Operating Systems Principles, SOSP, ACM, 2015.
[37] O. Padon, G. Losa, M. Sagiv, S. Shoham, Paxos made EPR: decidable reasoning about distributed protocols, Proc. ACM Program. Lang. 1 (2017) 108:1–108:31, https://doi.org/10.1145/3140568.
[38] P. Fonseca, K. Zhang, X. Wang, A. Krishnamurthy, An empirical study on the correctness of formally verified distributed systems, in: Proc. of the Twelfth European Conf. on Computer Systems, ACM, 2017, pp. 328–343.

PAPER C

MBT/CPN: A TOOL FOR MODEL-BASED SOFTWARE TESTING OF DISTRIBUTED SYSTEMS PROTOCOLS USING COLOURED PETRI NETS

R. Wang, L. M. Kristensen, and V. Stolz

In Verification and Evaluation of Computer and Communication Systems, volume 11181 of *Lecture Notes in Computer Science*, pages 97–113, Springer International Publishing, 2018.

MBT/CPN: A Tool for Model-Based Software Testing of Distributed Systems Protocols Using Coloured Petri Nets

Rui Wang( ), Lars Michael Kristensen, and Volker Stolz

Department of Computing, Mathematics, and Physics,
Western Norway University of Applied Sciences, Bergen, Norway
{rwa,lmkr,vsto}@hvl.no

Abstract. Model-based testing is an approach to software testing based on generating test cases from models. The test cases are then executed against a system under test. Coloured Petri Nets (CPNs) have been widely used for modeling, validation, and verification of concurrent software systems, but their application for model-based testing has only been explored to a limited extent. The contribution of this paper is to present the MBT/CPN tool, implemented through CPN Tools, to support test case generation from CPN models. We illustrate the application of our approach by showing how it can be used for model-based testing of a Go implementation of the coordinator in a two-phase commit protocol. In addition, we report on experimental results for Go-based implementations of a distributed storage protocol and the Paxos distributed consensus protocol. The experiments demonstrate that the generated test cases yield a high statement coverage.

1 Introduction

Society is heavily dependent on software and software systems, and design- and implementation errors in software systems may render them unavailable and return erroneous results to their users. It is therefore important to develop techniques that can be used to ensure correct and stable operation of the software.

Model-based testing (MBT) [13] is a promising technique for using models of a system under test (SUT) and its environment to generate test cases for the system. MBT approaches and tools have been developed based on a variety of modeling formalisms, including flowcharts, decision tables, finite-state machines, Petri nets, state-charts, object-oriented models, and BPMN [6]. A test case usually consists of test input and expected output and can be executed against the SUT. The goal of MBT is validation and error-detection by finding observable differences between the behavior of an implementation and the intended behavior. Generally, MBT involves: (a) constructing a model of the SUT and its environment; (b) define test selection criteria for guiding the generation of test cases and the corresponding test oracle representing the ground-truth; (c) generation and execution of test cases; (d) comparison of the output from the test

M. F. Atig et al. (Eds.): VECoS 2018, LNCS 11181, pp. 97–113, 2018.
https://doi.org/10.1007/978-3-030-00359-3_7

case execution with the expected result from the test oracle. The component that performs (c) and (d) is known as a *test adapter* and uses the *test oracles* to determine whether a test has passed or failed.

Coloured Petri Nets (CPNs) [5] is a modeling language for distributed and concurrent systems combining Petri nets and the Standard ML programming language. Petri nets provide the primitives for modeling concurrency, synchronization and communication while Standard ML is used for modeling data. Construction and analysis of CPN models is supported by CPN Tools [2] which have been widely used for modeling and verifying models of complex systems for domains such as concurrent systems, communication protocols, and distributed algorithms [9]. Recently, work on automated code generation has also been done [8]. Comprehensive testing is an important task in the engineering of software, including the case of automated code generation, as it is seldom the case that the correctness of the model-to-text transformations and their implementation can be formally established. We have chosen CPNs as the foundation of our MBT approach due to its strong track record in modeling distributed systems, and the support for parametric models and compact modeling of data. Moreover, CPNs enables model validation prior to test case generation, and CPN Tools supports both simulation and state space exploration which is paramount for the development of our approach and for conducting practical experiments.

The main contribution of this paper is to present our approach to model-based testing using CPNs and the supporting MBT/CPN tool. MBT/CPN has been implemented on top of CPN Tools to support test case generation from CPN models. It has been developed as part of our ongoing research into MBT for quorum-based distributed systems [15]. The main idea underlying our approach is for the modeler to capture the observable input and output events (transitions) in a test case specification. A main facility of the tool is the uniform support for both state space and simulation-based test case generation. A second contribution of this paper is to experimentally evaluate the tool on a two-phase commit protocol implemented using the Go programming language, and to summarize experimental results from the application of MBT/CPN to a distributed storage protocol [15] and the Paxos distributed consensus protocol [14]. The distributed storage protocol and the Paxos protocol have both been implemented in the Go programming language [3] using a quorum-based distributed systems middleware [10]. These experiments show a high statement coverage and demonstrate in addition that the approach is able to detect programming errors via the generation and execution of unit and system tests.

The rest of this paper is organized as follows. Section 2 gives an overview of MBT/CPN and its software architecture. In Sect. 3 we introduce the two-phase commit transaction protocol that we use as a running example to present the features of MBT/CPN. Sections 4 and 5 explain how test case generation and test case execution are supported. Section 6 presents our experimental evaluation of MBT/CPN. In Sect. 7, we sum up conclusions and discuss related work. We assume that the reader is familiar with the basic concepts of Petri nets. The MBT/CPN tool is available via [11].

2 Tool Overview and Software Architecture

The MBT/CPN tool is implemented in the Standard ML programming language on top of the simulator of CPN Tools. In CPN models, Standard ML is used to define the data types of the model, to declare the colour set of places and the variables of transitions, for defining guards of transitions, and for the arc expressions appearing on the arcs connecting places and transitions. MBT/CPN provides the user with a set of Standard ML functions which can be invoked in order to perform test case generation.

Figure 1 gives an overview of the modules that constitute MBT/CPN and puts the tool into the context of model-based test case generation. The main outputs of the MBT/CPN tool are files containing Test Cases which can be read by a Reader of a test Adapter and executed by a Tester against the System Under Test (SUT). The Tester will provide the input events as stimuli to the SUT and compare the observed outputs from the SUT with the expected outputs.

The application of MBT/CPN requires the user to identify the *observable events* originating from occurrences of binding elements in the CPN model. A binding element is a pair consisting of a transition and an assignment of values to the variables of the transition. A binding element hence represents a mode in which a transition may be enabled and may occur. A test case is comprised of observable events where input events represent stimuli to the SUT and output events represent expected outputs. It is the expected outputs that are used as test oracles during test case execution to determine the overall test outcome.

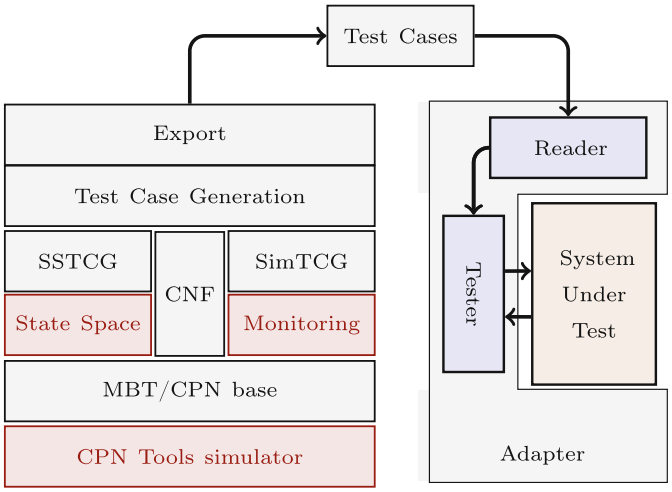


Fig. 1. Overview of MBT/CPN modules.

The MBT/CPN base module defines a generic colour set (data type) used to represent the observable events in test cases:

```
colset TCEvent = union InEvent:TCInEvent + OutEvent:TCOutEvent;
```

The definition of the colour sets `TCInEvent` and `TCOutEvent` depends on the SUT in terms of the events to be made observable. These must be defined by the user of the tool and can use the standard colour set constructors in CPN Tools. The tool supports two approaches for extracting test cases from the model:

State-space based test case generation. This approach is based on generating the state space of the CPN model and extracting test cases by considering paths in the state space. This approach is implemented in the SSTCG module on top of the state space tool of CPN Tools.

```
signature TCSPEC = sig
  val detection   : Bind.Elem -> bool;
  val observation : Bind.Elem -> TCEvent list;
  val format      : TCEvent   -> string
end;
```

Fig. 2. Standard ML interface for test case specification.

Simulation-based test case generation. This approach is based on conducting a simulation of the CPN model and extracting the test case corresponding to the execution. This approach is implemented in the SIMTCG module on top of the simulation monitoring facilities of CPN Tools.

The state-space based approach works for finite-state models and is based on computing all reachable states and state changes of the CPN model. The simulation-based approach is based on running a set of simulations and extracting test cases from the corresponding set of executions. The advantage of the state-space based approach is that it covers all the possible executions of the CPN model which gives a high test coverage. However, if the CPN model is complex, the state-space based approach may be infeasible due to the state explosion problem. The advantage of the simulation-based approach over the state-space based approach is scalability when the complexity of the CPN model is high, while the disadvantage is potentially reduced test coverage.

The CNF (configuration) module is shared between the state space- and simulation-based test case generation. It supports configuring the output directories and naming of test cases, and configuration of a *test case generation specification*. The test case specification is used to specify the observable input and output events during test case generation and is comprised of a:

Detection function constituting a predicate on binding elements that evaluates to true for binding elements representing observable events.

Observation function which maps an observable binding element into an observable input or output event belonging to the `TCEvent` colour set.

Formatting function mapping observable events into a string representation which is used in order to export the test cases into files.

The test case specification is provided by the user implementing a Standard ML structure satisfying the `TCSPEC` signature (interface) shown in Fig. 2. The type `Bind.Elem` is an existing data type in CPN Tools representing binding elements. The observation function is specified to return a list of observable events to cater for the case where one might want to split a binding element into several observable events in the test case. We will give examples of detection and observation functions for the two-phase commit protocol example in Sect. 4.

The detection and observation functions are specified independently of whether simulation-based or state space-based test case generation is employed. This allows the input from the user to be specified in a uniform way, independently of which approach will be used for the test case generation. This makes it

```
signature TCGEN = sig
  val ss  : unit -> (TCEvent list) list;
  val sim : int -> (TCEvent list) list;
  val export : (TCEvent list) list -> unit
end;
```

Fig. 3. Standard ML interface for test case generation.

easy to switch between the two approaches. The tool invokes the detection function on each arc of the state space (occurring binding element in a simulation) to determine whether the corresponding event is observable, and if so, then the observation function will be invoked to map the corresponding binding element into an observable event. The `Export` module implements the export of the test cases into files and relies on the `CNF` module for persistence and naming.

When an implementation of the test case specification has been provided by the user, the MBT/CPN tool can be used to generate test cases. The primitives available for the user to control the test case generation are provided by the Test Case Generation module which implements the `TCGEN` interface (signature) partly shown in Fig. 3. The `ss` function is used for state-space based test case generation. The `sim` function is used for simulation-based test case generation and takes an integer as a parameter specifying the number of simulation runs that should be conducted to generate test cases. Both functions return a list of test cases, where each test case is comprised of a list of test case events (`TCEvent`). The `export` function is used for exporting the test cases into files according to the settings which the user provided via the CNF configuration module (Fig. 1).

3 Example: Two-Phase Commit Transaction Protocol

We use the two-phase commit transaction (TPC) protocol from [5] to explain the use of MBT/CPN. The CPN model is comprised of four hierarchically organized modules. Figure 4 shows the CPN module for the coordinator process and Fig. 5 shows the CPN module for the worker processes. Figure 6 shows model-based test case generation and exporting. Due to space limitations, we do not show the top-level CPN module and have also omitted the submodule of the CollectVotes substitution transition in Fig. 4. Each port place (place drawn with a double border) in the coordinator module is linked via so-called port-socket assignments to the accordingly named place in the workers module. The colour sets and variable used are shown in Fig. 7.

The coordinator starts by sending a message to each worker (transition SendCanCommit), asking whether the transaction can be committed or not. Each worker votes `Yes` or `No` (transition ReceiveCanCommit). The coordinator then collects each vote as modeled by the CollectVotes submodule of the CollectVotes substitution transition. Based on the collected votes, the coordinator sends back an `abort` or `commit` decision.

The coordinator will decide on commit if and only if all workers voted yes. The workers that voted yes then receive the decision (transition ReceiveDecision) and send back an acknowledgement. The coordinator then receives all acknowledgements (transition ReceiveAcknowledgement). After having executed the protocol, the place Completed will contain a token with colour `abort` or `commit` depending on whether the transaction was to be committed or not.

When presenting MBT/CPN in the remainder of this paper, we show how it can be used to generate test cases from the TPC CPN model. These can then be executed by a test adapter against an implementation of the coordinator process in the Go programming language. The workers module is used to obtain input events (stimuli) for the coordinator implementation, and the coordinator CPN module is used to obtain expected outputs (test oracles) which in turn determine whether a test is successful or not. In that respect, the CPN module of the coordinator serves as an abstract specification of the coordinator process against which the behavior of the implementation can be compared.

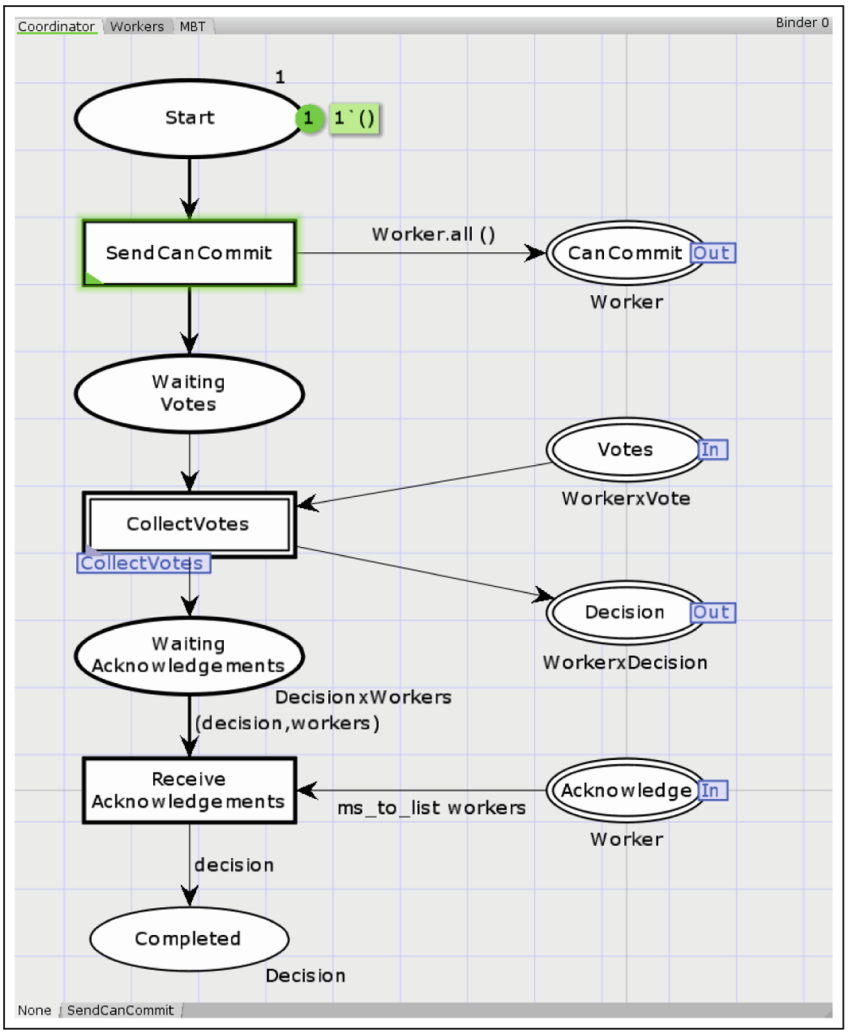


Fig. 4. MBT/CPN example in CPN Tools: Coordinator module.

4 Test Case Generation

The first step in using the MBT/CPN tool for test case generation is to extend the `TCEvent` base colour set by defining the colour sets `TCInEvent` and `TCOutEvent` according to the input and output events of the system that are to be observed. For the TPC protocol, we can define the input events to be the votes of the individual workers. The output events can be defined as the decisions sent to the individual workers and the overall decision as to whether the transaction is to be committed or aborted. Relying on the colour set definitions already in the CPN model (Fig. 7), this can be implemented as shown in Fig. 8. In the `TCOutEvent` colour set, `WDecision` is used for the decision sent to each worker while `SDecision` is used for the overall system decision.

For the TPC protocol, the input events corresponding to the votes sent by the workers can be obtained by considering occurrences of the ReceiveCanCommit transition (Fig. 5), while the output events can be obtained by considering the ReceiveDecision and ReceiveAcknowledgement transitions. This means that the

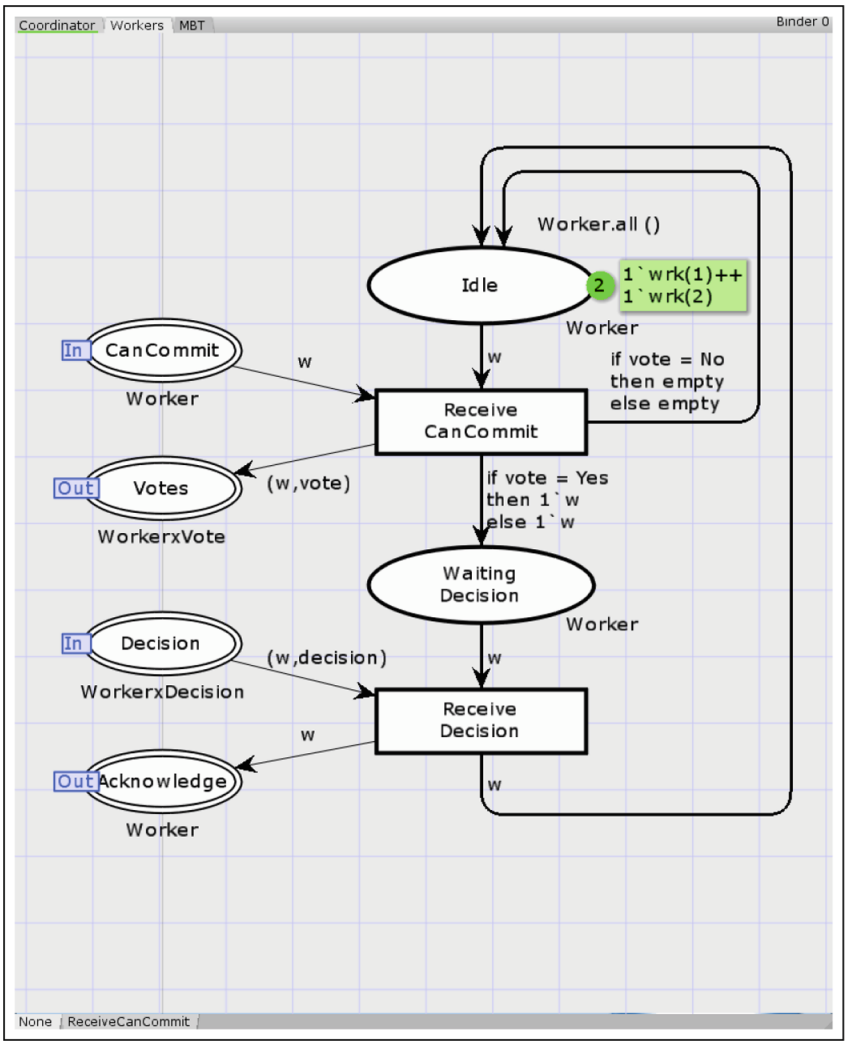


Fig. 5. MBT/CPN example in CPN Tools: Workers module.

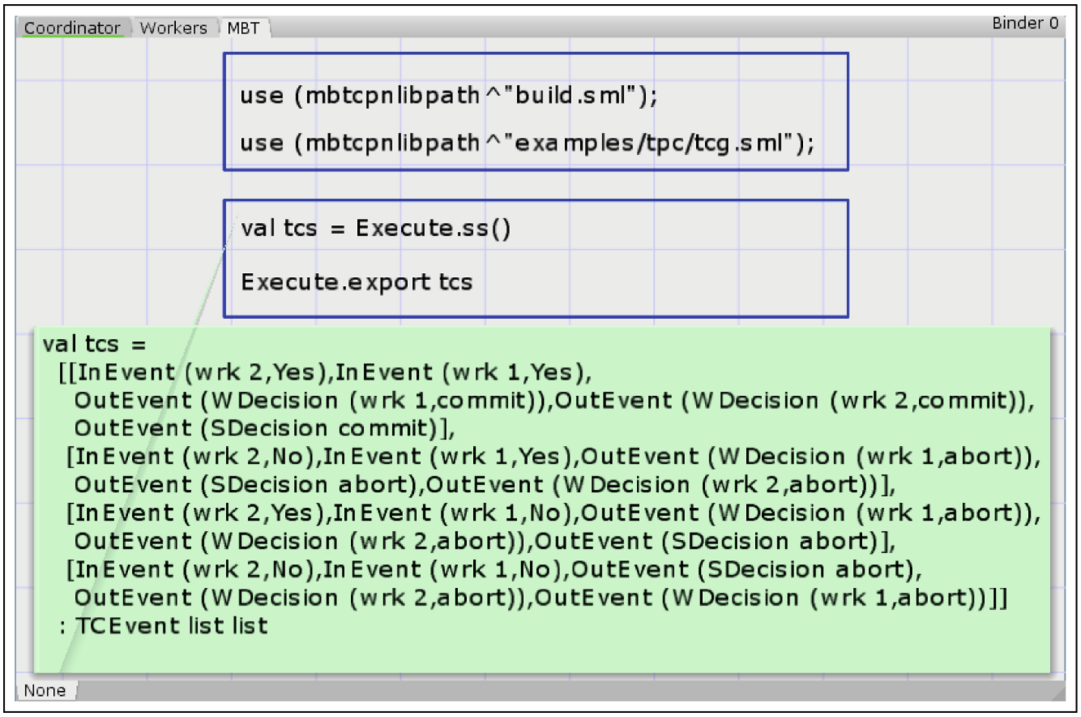


Fig. 6. MBT/CPN example in CPN Tools: Model-based test case generation and exporting.

```
val W = 2;
colset Worker  = index wrk with 1..W;     var w : Worker;
colset Workers = list Worker;             var workers : Workers;

colset Vote     = with Yes | No;          var vote : Vote;
colset Decision = with abort | commit;    var decision : Decision;

colset WorkerxVote      = product Worker * Vote;
colset WorkerxDecision  = product Worker * Decision;
```

Fig. 7. Colour set and variable declarations.

detection function for the TPC protocol must return true if and only if the occurrence of the binding element corresponds to one of the above-mentioned transitions. The implementation of the detection function is shown in Fig. 9.

The observation function maps binding elements into observable input and output events. For the TPC protocol this function can be implemented as in Fig. 10. The function accesses the values bound to the variables (`w`,`vote`, and `decision`) of the transitions and uses the constructors of the `TCEvent` and `TCOutEvent` data types to construct the observable events.

```
colset TCInEvent  = WorkerxVote;
colset TCOutEvent = union WDecision : WorkerxDecision +
                          SDecision : Decision;

colset TCEvent    = union InEvent  : TCInEvent +
                          OutEvent : TCOutEvent;
```

Fig. 8. Definitions of the colour sets `TCInEvent`, `TCOutEvent` and `TCEvent`.

```
fun detection (Bind.Workers'Receive_CanCommit _)  = true
  | detection (Bind.Workers'Receive_Decision _)  = true
  | detection (Bind.Coordinator'Receive_Acknowledgements _) = true
  | detection _ = false;
```

Fig. 9. The implementation of the detection function for the TPC protocol.

```
exception obsExn;
fun observation (Bind.Workers'Receive_CanCommit (_,{w,vote})) =
     [InEvent (w,vote)]
  | observation (Bind.Coordinator'Receive_Acknowledgements
     (_,{_,decision})) = [OutEvent (SDecision decision)]
  | observation (Bind.Workers'Receive_Decision (_,{w,decision})) =
     [OutEvent (WDecision (w,decision))]
  | observation _ = raise obsExn;
```

Fig. 10. The implementation of the observation function for the TPC protocol.

The MBT/CPN tool has built-in for exporting the test cases into an XML format. The use of XML makes it easy to reuse the test generator for systems under test implemented in different programming languages. The concrete XML format will depend on the observable events and hence the user needs to provide a `format` function as part of the test case generation specification that maps each observable event into a string representing an XML element. This function is typically implemented as a pattern match on the `TCEvent` data type. For the TPC protocol it would for instance map the `InEvent` corresponding to worker one (`wrk(1)`) voting `No` into the following XML element:

```
<Vote><WorkerID>1</WorkerID><VoteValue>0</VoteValue></Vote>
```

The complete formatting function for the TPC protocol is similar in complexity to the detection and the observation functions.

5 Test Case Execution

To perform model-based testing using the test cases generated by MBT/CPN, the developer (user) must implement a test Adapter as was shown in Fig. 1. The

implementation of the test adapter depends on the concrete SUT, but consists of the same overall components independently of the SUT. To illustrate how MBT/CPN test cases can be used, we outline how to implement a test adapter for a Go implementation of the coordinator process. The adapter consists of a Reader and a Tester. The implementation of the Reader (around 30 lines of code) is based on the *encoding/xml* package from the Go standard library, while the implementation of the Tester (around 80 lines of code) is based on *testing* packages of the Go standard library. Go's testing infrastructure allows us to run the `go test` command to execute the test cases and it provides pass/fail information for each test case. In addition, it provides information about code coverage. The full Go implementation of the adapter and also the coordinator SUT is available together with the MBT/CPN distribution [11].

The purpose of the reader is to read the XML files containing test cases and convert them into a representation which can be used by the tester. In this case, the *encoding/xml* package of the Go standard library supports the implementation of the Reader. The purpose of the tester is to provide input and read the output from the SUT according to the test case being executed. Hence, the tester serves as an intermediate between the test cases and the SUT. In this case, our coordinator SUT is implemented in Go, and the communication between the coordinator SUT and the tester is implemented using Go channels. The tester provides input to the coordinator SUT via the channels and implements the test oracles by comparing the values received with the expected output as specified in the test case. An important property of the tester implementation is that it is transparent to the coordinator SUT that it is interacting with the tester and not a real set of worker implementations.

The messages exchanged between the tester and the coordinator SUT are defined according to the mapping between the colour sets defined for messages in the CPN model (Fig. 7) and corresponding types in Go. Figure 11 shows the declarations of messages in Go for such communication which include `CanCommit`, `Vote`, `Decision` and `Ack` (Go code organized in two columns to save space).

The Go implementation of the coordinator SUT itself follows closely the CPN module of the coordinator (Fig. 4). Figure 12 shows the coordinator interface implemented in Go, which consists of methods for sending and delivering messages through channels. The method `Start` is the entry point of the coordinator which starts the coordinator's main control flow as a goroutine (thread). Within this loop, the coordinator receives incoming `Vote` and `Ack` messages through channels, delivered by the invocations of `DeliverVote` and `DeliverACK` methods, respectively. The coordinator invokes `CollectVotes` method to collect received `Vote` messages, and invoke `SendDecision` and `SendFinalDecision` methods to send `Decision` messages and a final `Decision` message.

```
type VoteEnum int
const (
   Yes VoteEnum = iota
   No
)
const (
   Commit DecisionEnum = iota
   Abort
)
type Decision struct {
   WorkerID      WorkerID
   DecisionValue DecisionEnum
}
type DecisionEnum int
type Vote struct {
   WorkerID  WorkerID
   VoteValue VoteEnum
}
type CanCommit struct {
   WorkerID WorkerID
}

type Ack struct {
   WorkerID WorkerID
}
```

Fig. 11. Message declarations in Go.

```
type Coordinator interface {
  Start(numOfWorker int, fdChannel chan DecisionEnum)
  SendCanCommit(cc CanCommit)
  DeliverVote(v Vote)
  CollectVotes(v Vote, votes []Vote) []Vote
  SendDecision(d Decision)
  DeliverACK(a Ack)
  SendFinalDecision(fdChannel chan DecisionEnum, fd DecisionEnum)
}
```

Fig. 12. Interface of the coordinator SUT in Go.

6 Experimental Evaluation

We report on experimental results on applying the MBT/CPN tool on the two-phase commit protocol with the coordinator as the system under test. In addition, we summarize experimental results obtained using our approach on two larger case studies: a distributed storage protocol and the Paxos consensus protocol. All three systems under test have been implemented in Go and the distributed storage and consensus protocol furthermore rely on the Gorums middleware [10]. The case studies illustrate the use of both simulation- and state space based test case generation. We use statement coverage of the system under test as the quantitative evaluation criteria of the test cases generated by our approach. Other criteria exist such as branch-, condition-, and path coverage, but these are currently not supported by the Go tool chain.

6.1 Two-Phase Commit Protocol

Table 1 gives experimental results from application of our approach to the two-phase commit protocol for different number of workers W. The Gen column specifies the approach used for test case generation (state spaces (SS) or simulation

(SIM)). The Size-Steps column specifies the size of the state space (nodes/arcs) and the number of simulation runs. The Test Cases column specifies the number of test case generated and the Time gives the total time (in second) used for test case generation (including state space generation and model simulation). Finally, the Coverage gives the statement coverage obtained for the coordinator implementation. The lines of code for the coordinator is around 120 lines.

Table 1. Experimental results for the two-phase commit protocol.

W	Gen	Size - Steps	Test Cases	Time	Coverage
2	SS	59/86	4	<1	94.7%
2	SIM	5	3	<1	84.2%
2	SIM	10	4	<1	94.7%
3	SS	357/614	8	<1	94.7%
3	SIM	10	4	<1	94.7%
3	SIM	20	8	<1	94.7%
4	SS	2,811/5,957	16	5	94.7%
4	SIM	50	13	<1	94.7%
4	SIM	100	16	<1	94.7%
5	SIM	100	31	<1	94.7%
5	SIM	200	32	<1	94.7%
10	SIM	5000	1,015	13	94.7%
10	SIM	10000	1,024	25	94.7%
15	SIM	10000	8,627	91	84.2%
15	SIM	20000	14,946	265	94.7%

For simulation-based test case generation, we stopped increasing the number of simulations when reaching the same number of test cases as obtained with state space based generation which represents the maximum number of test cases that can be obtained. It can be seen that as W increases more simulations are needed in order to reach the maximum number of test cases. In general, we recommend using state-space based test case generation whenever possible as it ensures coverage of all executions of the CPN model, and resort to simulation-based test case generation if the state space is too big to be generated with the available computing power. For the two-phase commit protocol we have not pursued state space based test case generation beyond four workers as it becomes quite time consuming. It can, however, be seen that simulation-based test case generation can easily handle configurations with 5, 10, and 15 workers demonstrating the scalability of simulation-based test case generation. The coverage results show that test cases generated based on state space and simulation based approaches can both reach 94.7%. The reason why the results do not reach 100% is that the coordinator contains error handling code, which is

not covered by the generated test cases, as any failures are not part of the model. The other coming two examples also have failures modeled explicitly. Further, the results also show that the statement coverage for both SIM-5 and SIM-10000 is 84.2%. This is a consequence of the simulation-based approach not covering all the possible executions of the CPN model in the absence of guided search. The longest time used for test case execution was approximately four hours (case SIM-20000) with more than 14,000 test cases.

6.2 Distributed Storage Protocol

The distributed storage protocol has been implemented by the Go language and Gorums framework. It is a single-writer, multi-reader distributed storage using read and write quorum calls and functions. The quorum calls and functions are abstractions provided by the Gorums framework/library. Clients can then invoke a write call with read calls concurrently and/or sequentially to access the distributed storage. By using our MBT/CPN tool, we have generated test cases based on the state-space based exploration to perform both system tests by invoking the read and write quorum calls concurrently and sequentially, and unit tests for quorum functions. The CPN model of the distributed storage makes it possible to generate system test cases for both successful scenarios and scenarios involving server failures and programming errors. We use a state-space based approach since the state space of the CPN testing model of the distributed storage protocol is relatively small. This is due to the fact that the CPN model describes the distributed storage system at a high level of abstraction which in turn means that we obtain all test cases without encountering state explosion.

Table 2 gives the experimental results obtained using different test drivers to invoke the read and/or write quorum calls concurrently and/or sequentially, without server failures included. The test drivers we have considered include: one read call (RD), one write call (WR), a read call followed by a write call (RD;WR), a write call followed by a read call (WR;RD), a read and a write call executed concurrently (WR||RD), a read and a write call executed concurrently and followed by a read call ((WR||RD);RD).

The results show that, for successful execution scenarios, the statement coverage for read (RD-QF) and write (WR-QF) quorum functions is 100% for both system and unit tests, as long as both read and write calls are involved. The statement coverage for read (RD-QC) and write (WR-QC) quorum calls is up to 84.4%. For the Gorums library as a whole, the statement coverage reaches 40.8%. The total number of lines of code for the system under test is approximately 2100 lines. The highest number of generated test cases for systems tests involving quorum calls is 6; the highest number of test cases for unit tests is 17. These test cases are generated within 2 s.

In addition to the successful scenarios, we has also considered to test the system under programming errors and server failures. We injected programming errors in the read and write quorum functions for the distributed storage such that the clients receive incorrectly replies from the storage system. The results show that our test adapter can capture injected errors by using generated test

Table 2. Experimental results for distributed storage protocol.

Test driver		Test case execution (coverage in percentage)				
		System			Unit	
ID	Name	Gorums library	QCs		QFs	
			RD	WR	RD	WR
S1	RD	24.6	84.4	0	100	0
S2	WR	24.6	0	84.4	0	100
S3	RD;WR	39.1	84.4	84.4	100	100
S4	WR;RD	40.8	84.4	84.4	100	100
S5	WR||RD	40.8	84.4	84.4	100	100
S6	(WR||RD);RD	40.8	84.4	84.4	100	100

cases from our MBT/CPN tool. For server failures scenario, we mainly test the fault tolerance of the distributed storage system. For example, a distributed storage system with three servers can tolerate one server failure. The test adapter we implemented can terminate one or more servers during the test case execution. We considered the S6 driver from Table 2 and created a scenario where S6 is executed first, then there is one or more server failures, and then S6 is repeated. The results for the scenario involving server failures show that the statement coverage for read (RD-QF) and write (WR-QF) quorum functions stay the same (100%) for both system and unit tests. The coverage for read (RD-QC) and write (WR-QC) quorum calls is increased from 84.4% to 96.7%. For the Gorums library as a whole, the statement coverage is increased from 40.8% to 52.3%.

6.3 Paxos Consensus Protocol

Paxos is a consensus protocol that can handle a group of server replicas to construct a replicated service, and ensure fault-tolerance. It is far more complex than the distributed storage system and the two-phase commit protocol. We have applied our MBT/CPN tool to validate a Go implementation of the single-decree Paxos. For such an implementation, each Paxos server replica implements a proposer, an acceptor, and a learner subsystem. In addition to these subsystems, the implementation also includes software components for failure and leader detection. Further, the communication and message handling between Paxos subsystems are implemented with quorum calls and functions (prepare, accept, and commit), which are abstractions from the Gorums framework. The total number of lines of code for the single-decree Paxos protocol is approximately 3890 lines.

The Paxos protocol is too complex for state space exploration, and we have therefore used simulation-based test case generation with up to 10 simulation runs. A summary of our experimental results is shown in Table 3. It shows the statement coverage obtained for the different Paxos subsystems, quorum calls

and functions. Note that the unit tests are only for the quorum functions. The total number of generated test cases for 3 and 5 replicas configurations, respectively are given below System tests and Unit tests in the table. The time used to generate test cases for each configuration is less than 10 s, and the time used to execute each test case is less than one minute.

Table 3. Experimental results for test case generation and execution.

Subsystem	Component	System tests	Unit tests
		15/38	74/424
Gorums library		51.8%	-
Paxos core	Proposer	97.4%	-
	Acceptor	100.0%	-
	Failure Detector	75.0%	-
	Leader Detector	91.4%	-
	Replica	91.4%	-
Quorum calls	Prepare	83.9%	-
	Accept	83.9%	-
	Commit	83.9%	-
Quorum functions	Prepare	100.0%	90.0%
	Accept	100.0%	85.7%

The results show that, for unit tests, the statement coverage of Prepare and Accept quorum functions reach 90% and 85.7%, respectively. For system tests, the statement coverage of Prepare, Accept and Commit quorum calls are up to 83.9%, respectively; the statement coverage for the Failure Detector and Leader Detector modules are 75.0% and 91.4%, respectively; the statement coverage of the Paxos replica module is up to 91.4%; for the Gorums library as a whole, the highest statement coverage is 51.8%.

7 Conclusions

The MBT/CPN tool augments the CPN Tools with facilities for model-based test case generation, and is based on the user identifying observable events formalized in a test case specification. As illustrated on the TPC protocol, this entails implementing a detection, observation, and formatting function which is applied by the tool during test case generation. An important feature of our approach is the uniform support for test case generation based on state spaces and simulation. We have shown by practical experiments on the TPC protocol, the distributed storage protocol, and the Paxos consensus protocol that we can obtain a high SUT code coverage and that our approach can be used to detect implementation errors.

The application of MBT in the context of CPNs have until now been limited. Xu [16] presents the Integration and System Test Automation (ITSA) tool which supports test code generation for languages such as Java, C/C++, and C♯ based on state spaces. To obtain concrete test cases with input data, the ITSA tool relies on a separate model implementation mapping. In contrast, we obtain the input data for the system under test and call directly from the data contained in the testing model. Tretmans et al. have presented the TorX [12] tool which is used to randomly generate test cases based on a walk through the state space. The test cases can be generated either offline or on-the-fly during the test execution. There is also an adapter component in TorX to translate the inputs to be readable by the system under test, and check the actual outputs from the system under test against expected outputs. Conformiq Qtroniq [4] can be used to derive functional test cases from a system model, and can generate test cases online or offline by using a symbolic execution algorithm. Such test cases then are mapped into the TTCN-3 format. The expected outputs can also be generated from the model. The Automatic Efficient Test Generation (AETG) [1] tool is aimed at efficient generation of test cases by decreasing the number of test data required for the input test space. However, the test oracles have to be furnished manually.

There are several interesting directions to further develop the MBT/CPN tool. Related to [17], one area is to provide a higher degree of automation when implementing the test adapter such that for instance the data types required in the adapter implementation can be automatically obtained. For simulation-based test case generation investigating how a search heuristic can be specified and synthesized is an important. Such heuristics will most likely require knowledge about the SUT implementation and its CPN model specification. For the latter, we are currently investigating how to measure so-called *Modified Condition/Decision Coverage*, which is prescribed e.g. in safety critical system development [7]. Another direction for future work is to investigate if the use of partial state spaces combined with a search heuristics can provide a fruitful middle ground between simulation-based and state space-based test case generation.

References

1. Cohen, D.M., Dalal, S.R., Fredman, M.L., Patton, G.C.: The AETG system: an approach to testing based on combinatorial design. IEEE Trans. Softw. Eng. **23**(7), 437–444 (1997)
2. CPN Tools. CPN Tools homepage. http://www.cpntools.org
3. Google Inc., The Go Programming Language. https://golang.org
4. Huima, A.: Implementing conformiq qtronic. In: Petrenko, A., Veanes, M., Tretmans, J., Grieskamp, W. (eds.) FATES/TestCom -2007. LNCS, vol. 4581, pp. 1–12. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73066-8_1
5. Jensen, K., Kristensen, L.: Coloured petri nets: a graphical language for modelling and validation of concurrent systems. Comm. ACM **58**(6), 61–70 (2015)
6. Jorgensen, P.: The Craft of Model-Based Testing. CRC Press, Boca Raton (2017)
7. Kelly, J.H., Dan, S.V., John, J.C., Leanna, K.R.: A Practical Tutorial on Modified Condition/Decision Coverage. Technical report (2001)

8. Kristensen, L.M., Veiset, V.: Transforming CPN models into code for TinyOS: a case study of the RPL protocol. In: Kordon, F., Moldt, D. (eds.) PETRI NETS 2016. LNCS, vol. 9698, pp. 135–154. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39086-4_10
9. Kristensen, L.M., Simonsen, K.I.F.: Applications of coloured petri nets for functional validation of protocol designs. In: Jensen, K., van der Aalst, W.M.P., Balbo, G., Koutny, M., Wolf, K. (eds.) Transactions on Petri Nets and Other Models of Concurrency VII. LNCS, vol. 7480, pp. 56–115. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38143-0_3
10. Lea, T.E., Jehl, L., Meling, H.: Towards new abstractions for implementing quorum-based systems. In: Proceedings of 37th IEEE International Conference on Distributed Computing Systems (ICDCS), pp. 2380–2385 (2017)
11. MBT/CPN. Repository, January 2018. https://github.com/selabhvl/mbtcpn.git
12. Tretmans, G., Brinksma, H.: TorX: automated model-based testing. In: Hartman, A., Dussa-Ziegler, K. (eds.) 1st European Conference on Model-Driven Software Engineering, vol. 12, pp. 31–43 (2003)
13. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Softw. Test. Verifi. Reliab. **22**, 297–312 (2012)
14. Wang, R., Kristensen, L., Meling, H., Stolz, V.: Automated test case generation for the paxos single-decree protocol using a coloured petri net model. J. Log. Algebraic Method. Programm. (JLAMP) (Submitted)
15. Wang, R., Kristensen, L., Meling, H., Stolz, V.: Application of model-based testing on a quorum-based distributed storage. In: Proceedings of PNSE 2017, CEUR Workshop Proceedings, vol. 1846, pp. 177–196 (2017)
16. Xu, D.: A tool for automated test code generation from high-level petri nets. In: Kristensen, L.M., Petrucci, L. (eds.) PETRI NETS 2011. LNCS, vol. 6709, pp. 308–317. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21834-7_17
17. Xu, D., Xu, W., Wong, W.E.: Automated test code generation from class state models. Int. J. Softw. Eng. Knowl. Eng. **19**(04), 599–623 (2009)

PAPER D

VISUALIZATION AND ABSTRACTIONS FOR EXECUTION PATHS IN MODEL-BASED SOFTWARE TESTING

R. Wang, C. Artho, L. M. Kristensen, and V. Stolz.

In Integrated Formal Methods, volume 11918 of *Lecture Notes in Computer Science*, pages 474–492, Springer International Publishing, 2019.

Visualization and Abstractions for Execution Paths in Model-Based Software Testing

Rui Wang[1( )], Cyrille Artho[2], Lars Michael Kristensen[1], and Volker Stolz[1]

[1] Department of Computing, Mathematics, and Physics,
Western Norway University of Applied Sciences, Bergen, Norway
artho@kth.se
[2] School of Computer Science and Communication,
KTH Royal Institute of Technology, Stockholm, Sweden
{rwa,lmkr,vsto}@hvl.no

Abstract. This paper presents a technique to measure and visualize execution-path coverage of test cases in the context of model-based software systems testing. Our technique provides visual feedback of the tests, their coverage, and their diversity. We provide two types of visualizations for path coverage based on so-called state-based graphs and path-based graphs. Our approach is implemented by extending the Modbat tool for model-based testing and experimentally evaluated on a collection of examples, including the ZooKeeper distributed coordination service. Our experimental results show that the state-based visualization is good at relating the tests to the model structure, while the path-based visualization shows distinct paths well, in particular linearly independent paths. Furthermore, our graph abstractions retain the characteristics of distinct execution paths, while removing some of the complexity of the graph.

1 Introduction

Software testing is a widely used, scalable and efficient technique to discover software defects [17]. However, generating sufficiently many and diverse test cases for a good coverage of the system under test (SUT) remains a challenge. *Model-based testing* (MBT) [21] addresses this problem by automating test-case generation based on deriving concrete test cases automatically from abstract (formal) models of the SUT. In addition to allowing automation, abstract test models are often easier to develop and maintain than low-level test scripts [20]. However, for models of complex systems, an exhaustive exploration of all possible tests is infeasible, and the decision of how many tests to generate is challenging.

Visualizing the degree to which tests have been executed can be helpful in this context: visualization can show if different parts of the model or SUT have been explored equally well [13], if there are redundancies in the tests [13], and if there are parts of the system that are hard to reach, e. g., due to preconditions that do not hold [3]. In this paper, we focus on the visualization of test paths on the test model, as this provides a higher level of abstraction than the SUT.

W. Ahrendt and S. L. Tapia Tarifa (Eds.): IFM 2019, LNCS 11918, pp. 474–492, 2019.
https://doi.org/10.1007/978-3-030-34968-4_26

The main contribution of this paper is to present a technique to capture and visualize execution paths of the model covered by test cases generated with MBT. Our approach records execution paths with a trie data-structure and visualizes them with the aid of lightweight abstractions as *state-based graphs* (SGs) and *path-based graphs* (PGs). These abstractions simplify the graphs and help us to deal with complexity in moderately large systems. The visual feedback provided by our technique is useful to understand to what degree the model and the SUT are executed by the generated test cases, and to understand execution traces and locate weaknesses in the coverage of the model. Being based on the state machine of the model, the state graph focuses on the behaviors of a system in relation to the test model. The path graph shows paths as transition sequences and eliminates crossing edges.

Our second contribution is to provide a path coverage visualizer based on the Modbat model-based API tester [2]. Our tool extends Modbat and enables the visualization of path coverage without requiring modifications of the models. Users of the tool can choose to visualize all execution paths in the SGs and PGs, or limit visualization to subgraphs of the SGs and PGs for models of large and complex systems. Our third contribution is an experimental evaluation on several model-based test suites. We analyze the number of executed paths against quantitative properties of the graphs. We show how edge thickness and colors help to visualize the frequency of transitions on executed paths, what kinds of paths have higher coverage than others, and what kinds of tests succeed or fail. We also compare the resulting SGs and PGs with *full state-based graphs* (FSGs) and *full path-based graphs* (FPGs). The FSGs and FPGs are the graphs without applying abstractions; they are used only in this paper for comparison with the SGs and PGs. We show that our abstraction technique helps to reduce the number of nodes and edges to get concise and abstracted graphs.

The rest of this paper is organized as follows. Section 2 gives background on extended finite state machines and Modbat. In Sect. 3, we give our definition of execution paths and the trie data structure used for their representation. In Sect. 4, we introduce our approach for the path coverage visualization. In Sect. 5, we present the two types of graphs and the associated abstractions used. Section 6 presents our experimental evaluation of the path coverage visualizer tool and analyzes path coverage of selected test examples. In Sect. 7, we discuss related work, and in Sect. 8 we sum up conclusions and discuss future work.

2 Extended Finite State Machines and Modbat

We use extended finite state machines (EFSMs) as the theoretical foundation for our models and adapt the classical definition [6] to better suit its implementation as an embedded language, and several extensions that Modbat [2] defines.

Definition 1 (Extended Finite State Machine). *An* extended finite state machine *is a tuple* $M = (S, s_0, V, A, T)$ *such that:*

– S *is a finite set of states, including an initial state* s_0*.*

- $V = V_1 \times \ldots \times V_n$ *is an n-dimensional vector space representing the set of values for variables.*
- *A is a finite set of actions* $A : V \to (V, R)$*, where* $res \in R$ *denotes the* result *of an action, which is either* successful, failed, backtracked, *or* exceptional. *A successful action allows a test case to continue; a failed action constitutes a test failure and terminates the current test; a backtracked action corresponds to the case where the enabling function of a transition is* false *[6]; exceptional results are defined as such by user-defined predicates that are evaluated at run-time, and cover the non-deterministic behavior of the SUT. We denote by* $Exc \subset R$ *the set of all possible exceptional outcomes.*
- *T is a transition relation* $T : S \times A \times S \times E$*; for a transition* $t \in T$ *we denote the left-side (origin) state by* $s_{origin}(t)$ *and the right-side (destination) state by* $s_{dest}(t)$*, and use the shorthand* $s_{origin} \to s_{dest}$ *if the action is uniquely defined. A transition includes a possible empty mapping* $E : Exc \to S$*, which maps exceptional results to a new destination state.*

Compared to the traditional definition of an EFSM [6], we merge the enabling and update functions into a single action $\alpha \in A$, and handle inputs and outputs inside the action. Actions deal with preconditions, inputs, executing test actions on the SUT, and its outputs. An action may also include assertions; a failed assertion causes the current test case to fail. Finally, transitions support non-deterministic outcomes in our definition.

Modbat. Modbat is a model-based testing tool aimed at performing online testing on state-based systems [2]. Test models in Modbat are expressed as EFSMs in a domain-specific language based on Scala [18]. The model variables can be arbitrarily complex data structures. Actions can update the variables, pass them as part of calls to the SUT, and use them in test oracles.

Figure 1(left) shows the ChooseTest model that we will use as a simple running example to introduce the basic concepts of Modbat and our approach to execution path visualization and abstraction. A valid execution path in a Modbat model starts from the initial state and consists of a sequence of transitions. The first declared state automatically constitutes the initial state. Transitions are declared with a concise syntax: "*origin*" → "*dest*" := {*action*}. The ChooseTest model in Fig. 1 consists of three states: "*ok*", "*end*", and "*err*". It also uses `require` in the action part as a precondition to check if a call to the random function `choose` returns `0` (10% chance). Only in that case is the transition from "*ok*" to "*err*" enabled. Function `assert` is then used to check if a subsequent call to `choose` returns non-zero. If `0` is returned (10% chance), the assertion fails. Thus, transition "*ok*" → "*err*" is rarely enabled; and if enabled, it fails only infrequently.

Choices. Modbat supports two kinds of *choices*: (1) Before a transition is executed, the choice of the next transition is available. (2) Within an action, choices

```
class ChooseTest extends Model {
  "ok" -> "ok" := skip
  "ok" -> "end" := skip
  "ok" -> "err" := {
    require(choose(0, 10) == 0)
    assert(choose(0, 10) != 0)
  }
}
```

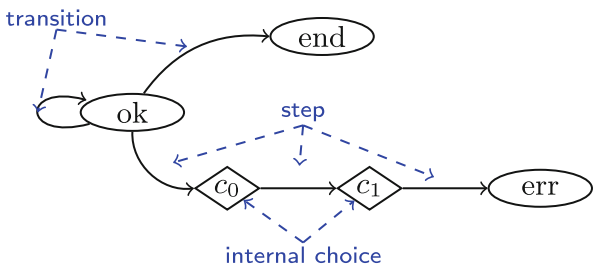


Fig. 1. Model ChooseTest (left) with steps and internal choices (right).

can be made on parameters that can be used as inputs to the SUT or for computations inside the action. The latter are *internal choices*, which can be choices over a finite set of numbers or functions. These choices are obtained in Modbat by calling the function `choose`. In our example, the action in transition "*ok*" to "*err*" has two internal choices shown as c_0 and c_1 in Fig. 1 (right).

Transitions and Steps. We divide an action into smaller *steps* to distinguish choices between transitions from internal choices inside an action. A step is a maximal-length sequence of statements inside an action that does not contain any choices. Our definition of choices corresponds to the semantics of Modbat, but also that of other tools, such as Java Pathfinder [22], a tool to analyze concurrent software that may also contain non-deterministic choices on inputs.

Action Results. Modbat actions (which execute code related to transitions) have four possible outcomes: successful, backtracked, failed, or exceptional. A successful action allows a test case to continue with another transition, if available. An action is *backtracked* and resets the transition to its original state if any of its preconditions are violated. An action *fails* if an assertion is violated, if an unexpected exception occurs, or if an expected exception does not occur. In our example, the action of transition "*ok*" to "*err*" is backtracked if the `require`-statement in the action evaluates to `false`, and the action *fails* if the `assert`-statement evaluates to `false`. *Exceptional results* are defined by custom predicates that may override the destination state ($s_{dest}(t)$) of a transition t; see above. If no precondition or assertion is violated, and no exceptional result occurs, the action is *successful*.

3 Execution Paths and Representation

Path coverage concerns a sequence of branch decisions instead of only one branch at a time. It is a stronger measurement than branch coverage, since it considers combinations of branch decisions (or statements) with other branch decisions (or statements), which may not have been tested according to the plain branch or statement coverage [16]. It is hard to reach 100% path coverage, as the number of execution paths usually increases exponentially with each additional branch or cycle [15].

A finite *execution path* is a sequence of transitions starting from the initial state and leading to a terminal state. A *terminal state* in our case is a state without outgoing transitions, or a state after a test failed. We denote by $S_{terminal}$ the set of terminal states.

Definition 2 (Execution Path). *Let $M = (S, s_0, V, A, T)$ be an EFSM. A* finite execution path *p of M is a sequence of transitions, which constitute a path $p = t_0 t_1 \dots t_n, \quad t_n \in T$, such that $s_{origin}(t_0) = s_0$, the origin and destination states are linked: $\forall i, 0 < i \leq n, s_{origin}(t_i) = s_{dest}(t_{i-1})$, and $s_{dest}(t_n) \in S_{terminal}$.*

We first represent the executed paths in a data structure based on the transitions executed by the generated test cases, and then use this to visualize path coverage of a test suite in the form of state-based and path-based graphs.

We record the path executed by each test case in a *trie* [5]. A trie is a prefix tree data structure where all the descendants of a node in the trie have a common prefix. Each trie node n stores information related to an executed transition, including the following: t (executed transition); *ti* (transition information); *trc* (transition repetition counter) to count the number of times transition t has been executed repeatedly without any other transitions executing in between during a test-case execution, with a value of 1 equalling no repetition; *tpc* (transition path counter) to count the number of paths that have this transition t executed *trc* times in a test suite; *Ch*, the set of children of node n; and *lf*, a Boolean variable to decide if the current node is a leaf of the tree. The transition information *ti* consists of the $s_{origin}(t)$ and $s_{dest}(t)$ states of the transition, a transition identifier *tid*, a counter *cnt* to count the number of times this transition is executed in a path, an action result *res*, which could be successful, backtracked, or failed, and sequences of transition-internal choices C for modeling non-determinism.

As an example, consider a test suite consisting of three execution paths: $p_0 = [a \to b, b \to b, b \to c, c \to d]$, $p_1 = [a \to b, b \to b, b \to b, b \to c, c \to d]$, and $p_2 = [a \to b, b \to b, b \to e]$, where a, b, c, d, and e are states. These execution paths can be represented by the trie data structure shown in Fig. 2 where the node labeled root represents the root of the trie. Note that this data structure is not a direct visual representation of the paths and it is not the trie data structure that we eventually visualize in our approach. Each non-root node in the trie in Fig. 2 has been labeled with the transition it represents. As an example, node 1 represents the transition $a \to b$ and node 2 represents the transition $b \to b$. This reflects that all the three execution paths stored in the trie have $a \to b$ followed by $b \to b$ as a (common) prefix. Each non-root node also has a label representing the transition counters associated with the node. For the transition counters, the value before the colon is *trc* (transition repetition counter), while the value after the colon is *tpc* (transition path counter). For example, the transition $b \to b$ associated with node 2 has been taken three times in total. Two paths, (p_0 and p_2) execute this transition once (label trc=1:tpc=2), while one path p_1 executes it twice (label trc=2:tpc=1). A parent node and a child node in the trie are connected by a mapping $\langle tid, res \rangle \mapsto n$ in each node which associates a transition identifier and action result (res) with a child (destination) node n.

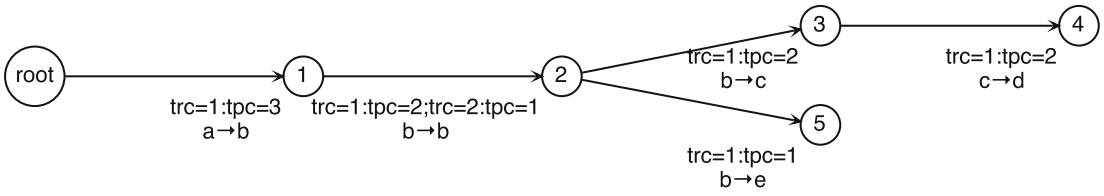
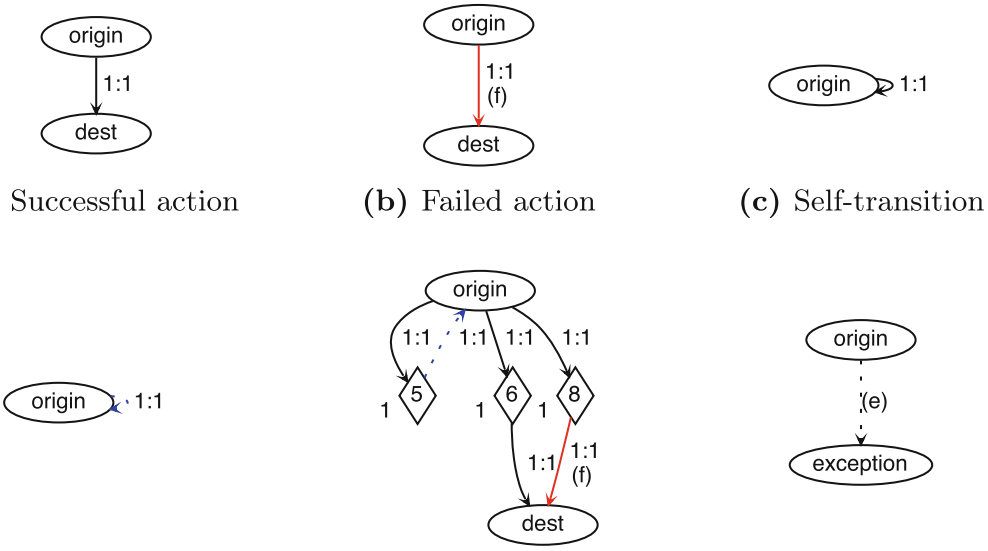


Fig. 2. Example trie data structure representing three executed paths.



(a) Successful action **(b)** Failed action **(c)** Self-transition

(d) Backtracked transition **(e)** Action with choices **(f)** Exception result

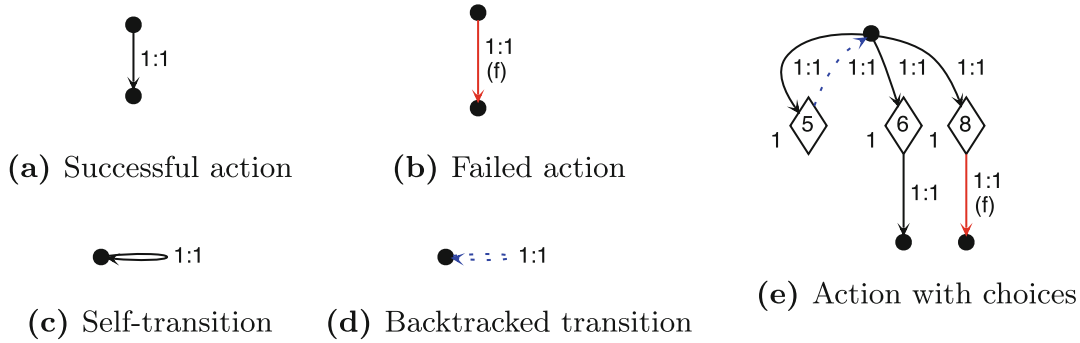
Fig. 3. Basic visualization elements of the state-based graphs (SGs). (Color figure online)

4 Path Coverage Visualization

Our path coverage visualizer can produce two types of directed graphs: state-based graphs (SGs) and path-based graphs (PGs). These two types of graphs are produced based on the data stored in the trie data structure representing the executed paths of the testing model. Figures 3 and 4 illustrate the basic visualization elements of the SGs and PGs, respectively, with the help of the DOT Language [9], which can be used to create graphs with Graphviz tools [4].

The SGs and PGs have common node and edge styles (shape, color and thickness) to indicate different features of the path- execution coverage visualization.

Node Styles. We use three types of node shapes in the graphs for path coverage visualization. Elliptical nodes ⬭ represent states in the SG as shown in Fig. 3. Point nodes • represent the connections between transitions/steps in the PG as shown in Fig. 4. Diamond nodes ◊ visualize internal choices in both the SGs and PGs as shown in Figs. 3e and 4e. Each diamond node has a value inside indicates the chosen value. There is also an optional counter value label aside each diamond node to show how many times this choice has been taken. The edge labels of the format $n : m$ will be discussed later.



(a) Successful action **(b)** Failed action **(c)** Self-transition **(d)** Backtracked transition **(e)** Action with choices

Fig. 4. Basic visualization elements of the path-based graphs (PGs). (Color figure online)

Edge Styles. A directed edge in both the SGs and PGs represents an executed transition and its related information as stored in the trie structure. We distinguish different kinds of edges based on the action results, using shape and color styles. Black solid edges are used to represent successful transitions (Figs. 3a and 4a). Blue dotted edges are used to visualize backtracked transitions (Figs. 3d and 4d). Red solid edges labeled (f) are used to visualize failed transitions (shown in Figs. 3b and 4b). Black solid loops represent self-transitions (Figs. 3c and 4c) and are used when $s_{dest}(t)$ and $s_{origin}(t)$ of a transition t are the same state. Black dotted edges labeled (e) are used to represent exceptional results for the SG (shown in Fig. 3f). This allows the visualization to distinguish between the normal destination state $s_{dest}(t)$ and the exception destination state. For the PG, this kind of edge is ignored by merging the point nodes of $s_{origin}(t)$ and the exception destination state of a transition t into one point node. If a transition t consists of multiple *steps* (Figs. 3e and 4e), we only apply the edge styles to the last step edge which connects to $s_{dest}(t)$, while other step edges use a black solid style.

Each edge may have a label for additional information, such as transition identifier *tid*, and values of the counters *trc* and *tpc*. Here we use the format *trc* : *tpc*. It is optional to show these labels. For example, in both Figs. 3 and 4, the values of counters are all 1 : 1 indicates that each transition in a test case is executed only once without any repetitions, and there is only one path that has this transition executed.

The thickness of an edge indicates how frequently a transition is executed for the entire test suite. The thicker an edge is, the more frequently is its transition executed. Let *nTests* be the total number of executed test cases. Then, the thickness of an edge is given by $\ln(\frac{\sum count*100}{nTests} + 1)$, where the value of *count* is the *tpc* value of a transition in each path if there are no internal choices for this transition. If a transition has internal choices, then we use the value of the counter for each internal choice as the value of *count*. Since we merge edges in the graphs corresponding to the same transitions or the same choices from different paths, we then compute the sum of values of *counts* obtained for the transition or choice.

5 State-Based and Path-Based Graphs

We now present the details of the state-based (SG) and path-based (PG) graphs with abstractions that form the foundation of our visualization approach. These abstractions underly the reduced representation of the execution paths.

McCabe [12] proposed basic path testing and gave the definition of a *linearly independent path*. A linearly independent path is any path through a program that contains at least one new edge which is not included in any other linearly independent paths. A subpath q of an execution path p is a subsequence of p (possibly p itself), and an execution path p traverses a subpath q if q is a subsequence of p. In this paper, for the visualization of execution paths, we merge subpaths from different linearly independent paths in both SG and PG with the aid of the trie data structure.

5.1 State-Based Graphs

An SG is a directed graph $SG = (N_s, E_t)$, where $N_s \equiv \{n_{s_0}, n_{s_1}, \ldots, n_{s_i}\}$ is a set of nodes including both elliptical nodes representing states with their names and diamond nodes representing internal choices with their values as discussed in Sect. 4. Elliptical nodes use the name of their state as node identifier; diamond nodes are identified by a tuple $\langle v, cn \rangle$, where v is the value of the choice, and cn is an integer number starting from 1 and increasing with the number of diamond nodes. $E_t \equiv \{e_{t_0}, e_{t_1}, \ldots, e_{t_i}\}$ is a set of directed edges representing both transitions and steps. These edges connect nodes according to node identifiers.

An SG is an abstracted graph of the unabstracted full state-based graph (FSG). An FSG may have redundant edges representing the same transition/step between two states; it may also contain choices with the same choice value appearing more than once. These situations, in general, contribute to making the FSG large, complex and difficult to analyze, especially for large and complex systems. Note that the FSG is only used by us to show its complexity in this paper for comparison with the SG. The FSG for the ChooseTest model (discussed in Sect. 3) is already very dense after only 100 test cases (see Fig. 5).

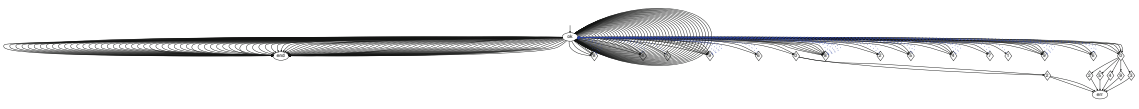


Fig. 5. FSG for 100 test cases of ChooseTest.

In order to reduce the complexity of graphs such as Fig. 5, we abstract the FSG to get the SG, and use edge thickness to indicate the frequency of transitions in the executed paths. We use the ChooseTest model with 1000 executed test cases as an example to show how the SG is obtained in four abstraction steps:

1. *Merge edges of subpaths*: the trie data structure is used to merge subpaths of linearly independent paths when storing transitions in the trie. As discussed for Fig. 2, transition $a \rightarrow b$ followed by $b \rightarrow b$ is a (common) prefix for all the three execution paths p_0, p_1 and p_2. In other words, all these three execution paths traverse the subpaths $a \rightarrow b$ and $b \rightarrow b$. Therefore, to obtain the SG, edges representing transition $a \rightarrow b$ and $b \rightarrow b$ from three execution paths are merged into one edge by the trie data structure. We then use an edge label of the form "*trc* : *tpc*" to show how a transition represented by this edge is executed. (Here, we do not show edge labels due to space limitations.) After merging edges of subpaths, we get only linearly independent paths in the graph. Fig. 6 shows the graph of the ChooseTest model after merging subpaths. There are seven linearly independent paths: $p_0 = [ok \rightarrow end]$, $p_1 = [ok \rightarrow ok, ok \rightarrow end]$, $p_2 = [ok \rightarrow ok, ok \rightarrow err(backtracked), ok \rightarrow end]$, $p_3 = [ok \rightarrow ok, ok \rightarrow err]$, $p_4 = [ok \rightarrow err(backtracked), ok \rightarrow end]$, $p_5 = [ok \rightarrow err(failed)]$ and $p_6 = [ok \rightarrow err]$.
2. *Merge edges of linearly independent paths*: from Fig. 6, it can be noticed that after merging edges of subpaths, the graph may still have redundant edges between two states that represent the same transition with the same action result from different linearly independent paths. For example, there are four edges between the "ok" and "end" states, from four linearly independent paths: p_0, p_1, p_2 and p_4. We merge such edges into one single edge. We also aggregate the path coverage counts. The aggregated counts can be shown as an optional edge label on the form "*trc* : *tpc*", using ";" as the separator, e. g., "$1 : 304; 1 : 158; 1 : 177; 1 : 290$" for the edge between the "ok" and "end" states after merging p_0, p_1, p_2 and p_4.
3. *Merge internal choice nodes*: internal choice nodes of a transition are merged in two ways. First, based on Step 1, when storing transitions in the trie, each transition has recorded choice lists; we merge choice nodes from different choice lists if these choice nodes have the same choice value and they are a (common) prefix of choice lists. For example, for choice lists $[0, 1, 3]$ and $[0, 1, 3]$ (0, 1, 2, 3 are choice values), we notice that these two choice lists both have choice nodes with value 0 and 1, and they are a (common) prefix for these two lists. We then merge choice nodes with value 0 and 1 to become one choice node, respectively, when storing transitions in the trie. Second, if there are still choice nodes of a transition from different linearly independent paths, with the same value appearing more than once, such as choices in Fig. 6, then we merge them into one choice node during Step 5.1. For both approaches, we get the result of the sum of the values of counters of merged choice nodes. This result denotes the total number of times a choice value appears in the SG, and it can then be shown in addition to the outcome of the choice on the label of the final choice node after merging. Note that to avoid visual clutter, we elide showing the target state for backtracked transitions.
4. *Merge loop edges*: loop edges represent self-transition loops and backtracked transitions; they are merged if they represent the same transition with the same action result.

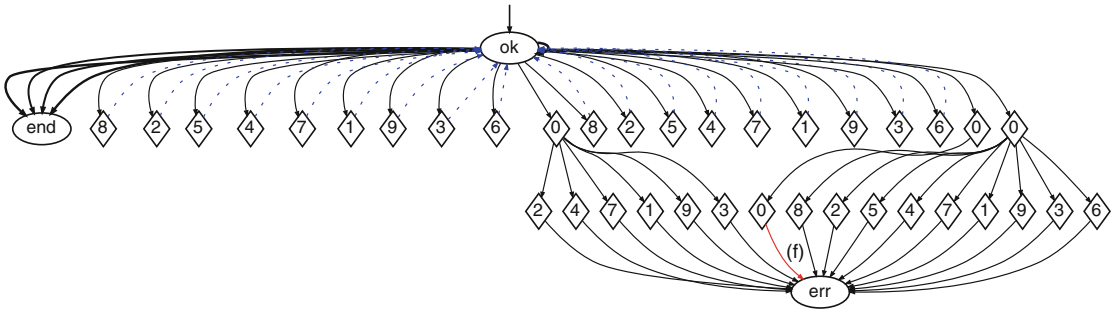


Fig. 6. The graph for 1000 test cases of ChooseTest after merging subpaths.

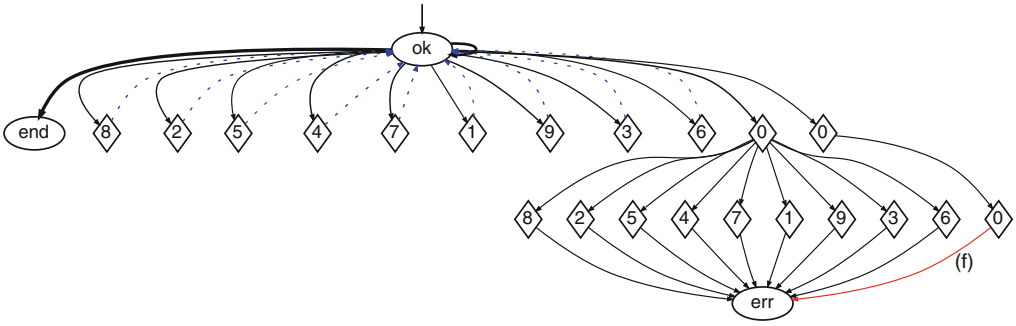


Fig. 7. SG for 1000 test cases of ChooseTest with all abstractions applied.

Figure 7 illustrates the final SG with all abstractions after 1000 test cases. One characteristic of the SG is that it is a concrete instance of its underlying state machine graph. The EFSM shows potential transitions, whereas the SG shows the actually executed steps of actions, and internal choices for non-determinism. For example, Fig. 7 is a concretization of the state machine shown in Fig. 1. As shown in Fig. 1, the transition "*ok*" → "*err*" has a precondition with an internal choice over values 0 to 9 (see Fig. 7). Only choice 0 enables this transition; the transition is backtracked the original state "*ok*" otherwise, as shown with the blue dotted edges. If the transition is enabled by a successful choice with value 0, the assertion, which is another internal choice that fails only for value 0 out of 0 to 9, is executed; its failure is shown by the red solid (failing) edge in Fig. 7.

5.2 Path-Based Graphs

The PG is a directed graph $PG = (N_p, E_t)$, where $N_p \equiv \{n_{p_0}, n_{p_1}, \ldots, n_{p_i}\}$ is a set of nodes, including point nodes representing connections between transitions and diamond nodes representing internal choices with their values (see Sect. 4); and $E_t \equiv \{e_{t_0}, e_{t_1}, \ldots, e_{t_i}\}$ is a set of directed edges representing both transitions and steps. They connect nodes using the identifiers of nodes.

The nodes in PG in contrast to SG do not correspond to states in the EFSM; instead, each node corresponds to a step in a linear independent path through the EFSM. Therefore, each point node is allocated a point node identifier *pn*, an integer starting from 0 (we elide the label in the diagrams here). The value of *pn*

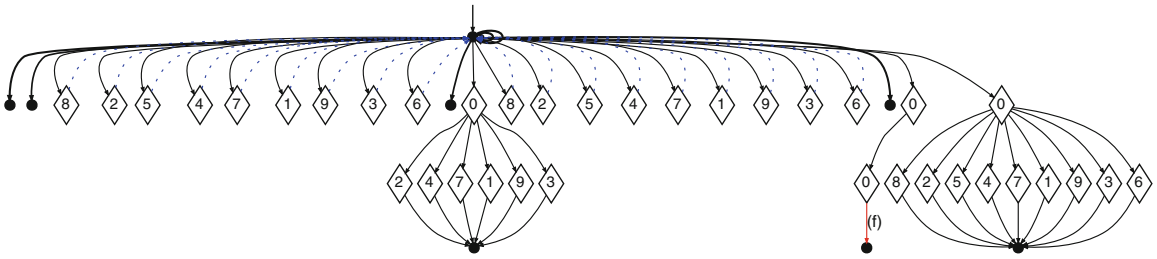


Fig. 8. Path-based graph for 1000 test cases of the ChooseTest model.

increases with the number of point nodes. Each diamond node is identified by a tuple $\langle v, cn \rangle$, similar to the diamond nodes in the SG. For the edges representing transitions, they are connected by point nodes according to their identifiers, which results in constructing paths one by one. All constructed paths start with the same initial point node and end in different final point nodes. The number of constructed paths in the PG indicates the number of linearly independent paths.

An PG is an abstracted graph of the full state-based graph (FPG) without any abstractions. (Here, we do not show the FPG of the ChooseTest model due to space limitations.) As was the case for SG, we apply abstractions to reduce the complexity of the FPG to obtain the PG and use thickness to indicate the frequency of transitions taken by executed paths. The reduction is based on three abstractions:

1. Merge edges of subpaths using the same approach as for step 1 of the SG.
2. Merge internal choice nodes with the first approach of step 3 used to merge choice nodes for the SG.
3. Merge loop edges representing self-transition loops and backtracked transitions as for the SG.

Unlike for the SG, we do not merge edges of linearly independent paths and choice nodes from different linearly independent paths for the PG, since our goal for the PG is to show linearly independent paths after applying the abstractions.

Figure 8 shows the abstracted PG for the ChooseTest model. It can be seen that there are seven black final point nodes from seven paths, which indicate that seven linearly independent paths have been executed. The information about the number of linearly independent paths is one characteristic of the PG, and this information is not easy to derive from the SG shown in Fig. 7.

5.3 User-Defined Search Function

As the SG and PG graphs might become unwieldy for complex testing models, the user can specify a *selection function* to limit the visualization to a subgraph. After completion of the tests, the user can filter the graph into a subgraph by providing a query in the form of a quadruple $\langle tid, res, l, ptid \rangle$ to locate a recorded transition in the trie data structure, where *tid* is the transition identifier for the transition that the user wants to locate; *res* is the action result of this transition;

l is the level of this transition in the trie; *ptid* is the transition identifier for this transition's parent in the trie. With this selection function, users can select a subtrie to generate both SG and PG with the corresponding root node in lieu of an interactive user-interface. It should be noted that this projection only affects visualization, and not the number of executed tests.

6 Experimental Evaluation

We have applied and evaluated our path coverage visualization approach on a collection of Modbat models. The list of models includes the Java server socket implementation, the coordinator of a two-phase commit protocol, the Java array list and linked list implementation, and ZooKeeper [11]. The array and linked list models, as well as the ZooKeeper model, consist of several parallel EFSMs, which are executed in an interleaving way [2].

Table 1 summarizes the results. For each Modbat model, we have considered configurations with 10, 100, 200, 500 and 1000 randomly generated test cases. The table first lists the statistics reported by Modbat: the number of states (S) and transitions (T) covered for each model (including their percentage), and the number of test cases (TC) and failed test cases (FC). The second part of the table shows the metrics of the graphs we generate. For both SGs and PGs, we list: the total number of Nodes (including both state nodes and choice nodes); the total numbers of Edges (E), the number of failed edges (FE), and loops (L). In addition to these graph metrics, for the PGs, our path coverage visualizer calculates the numbers of linearly independent paths (LIP), the longest paths (LP), the shortest paths (SP), the average lengths of paths (AVE), and the corresponding standard deviation (SD).

In Table 1, when comparing the results of the SG and PG obtained from all the models, we can see that for any increase in the number of test cases by going from 10 to 1000, the SG has a smaller number of nodes and edges than the PG. This shows that the SG is constructed in a more abstract way than the PG and is useful for giving an overview of the behavior. For the PG, although there are more nodes and edges in the graph compared to the SG, we can directly see the information about the number of linearly independent paths (LIP column in Table), so that we know how execution paths are constructed and executed from the sequences of transitions executed. This information cannot be easily seen from the SG.

In addition, the results in Table 1 indicate what degree the models are executed by the generated test cases. For example, for the coordinator model, the numbers of nodes and edges in both the PG and SG do not increase after 100 test cases are executed, and there are no failed edges. This gives us confidence about how well this model is explored by the tests. The same situation occurs for the array and linked list models. For the Java server socket and Zookeeper models, the number of failed edges for each model keeps increasing with more tests. This indicates that for these kinds of complex models, there are parts that are hard to reach and explore, so there might be a need to increase the number

Table 1. Experimental results for the Modbat models.

Model	S	T	TC	FC	Path-based (PG)									State-based (SG)			
					Nodes	Edges			Paths					Nodes	Edges		
						E	FE	L	LIP	LP	SP	AVE	SD		E	FE	L
JavaNio ServerSocket	7/ 7 (100%)	17/17 (100%)	10	2	57	79	1	17	8	14	3	9.25	4.18	9	23	1	6
			100	3	177	243	1	48	30	15	2	7.87	3.84	9	23	1	6
			200	8	363	528	4	111	53	29	2	9.68	6.24	10	25	1	7
			500	14	779	1147	8	247	105	29	2	10.51	5.29	11	27	1	8
			1000	28	1269	1904	15	439	168	29	2	10.80	4.79	11	27	1	8
Coordinator Test	7/ 7 (100%)	6/ 6 (100%)	10	0	17	20	0	0	1	6	6	6.00	0.00	17	20	0	0
			100	0	21	27	0	0	1	6	6	6.00	0.00	21	27	0	0
			200	0	21	27	0	0	1	6	6	6.00	0.00	21	27	0	0
			500	0	21	27	0	0	1	6	6	6.00	0.00	21	27	0	0
			1000	0	21	27	0	0	1	6	6	6.00	0.00	21	27	0	0
ArrayList Iterator ListIterator	1/ 1 (100%) 2/ 2 (100%) 2/ 2 (100%)	11/11 (100%) 5/11 (45%) 12/29 (41%)	10	0	174	542	0	276	6	99	12	58.17	38.75	34	85	0	38
ArrayList Iterator ListIterator	1/ 1 (100%) 2/ 2 (100%) 2/ 2 (100%)	11/11 (100%) 9/11 (81%) 13/29 (44%)	100	0	1171	3222	0	1571	75	181	2	23.93	29.74	102	216	0	94
ArrayList Iterator ListIterator	1/ 1 (100%) 2/ 2 (100%) 2/ 2 (100%)	11/11 (100%) 10/11 (90%) 17/29 (58%)	200	1	3369	10474	1	4848	138	181	2	45.35	47.46	204	423	1	184
ArrayList Iterator ListIterator	1/ 1 (100%) 2/ 2 (100%) 2/ 2 (100%)	11/11 (100%) 10/11 (90%) 25/29 (86%)	500	1	10438	29730	1	14024	319	181	2	48.96	43.55	467	955	1	417
ArrayList Iterator ListIterator	1/ 1 (100%) 2/ 2 (100%) 2/ 2 (100%)	11/11 (100%) 10/11 (90%) 27/29 (93%)	1000	14	29056	86871	1	40609	649	406	2	70.87	64.17	896	1812	1	815
LinkedList Iterator ListIterator	1/ 1 (100%) 2/ 2 (100%) 1/ 2 (50%)	18/19 (94%) 8/11 (72%) 5/29 (17%)	10	0	216	718	0	348	9	191	10	56.11	72.01	34	85	0	36
LinkedList Iterator ListIterator	1/ 1 (100%) 2/ 2 (100%) 1/ 2 (50%)	19/19 (100%) 9/11 (81%) 7/29 (24%)	100	0	1190	3348	0	1679	83	191	2	23.51	38.05	148	312	0	131
LinkedList Iterator ListIterator	1/ 1 (100%) 2/ 2 (100%) 2/ 2 (100%)	19/19 (100%) 9/11 (81%) 19/29 (65%)	200	0	6266	17140	0	7549	178	191	2	54.45	49.14	405	824	0	295
LinkedList Iterator ListIterator	1/ 1 (100%) 2/ 2 (100%) 2/ 2 (100%)	19/19 (100%) 9/11 (81%) 22/29 (75%)	500	0	15091	43303	0	19797	406	257	2	60.17	61.56	699	1413	0	522
LinkedList Iterator ListIterator	1/ 1 (100%) 2/ 2 (100%) 2/ 2 (100%)	19/19 (100%) 9/11 (81%) 24/29 (82%)	1000	0	39391	113155	0	52461	825	257	2	74.66	67.19	1404	2819	0	1083
ZKServer ZKClient	4/ 4 (100%) 9/13 (69%)	4/ 4 (100%) 28/54 (51%)	10	0	488	536	0	6	10	27	17	24.60	2.65	158	203	0	6
ZKServer ZKClient	4/ 4 (100%) 11/13 (84%)	4/ 4 (100%) 38/54 (70%)	100	7	4628	5160	7	76	98	31	4	22.57	5.99	862	1110	5	75
ZKServer ZKClient	4/ 4 (100%) 11/13 (84%)	4/ 4 (100%) 39/54 (72%)	200	9	9869	9869	9	138	197	31	4	22.88	5.67	1532	1964	5	135
ZKServer ZKClient	4/ 4 (100%) 11/13 (84%)	4/ 4 (100%) 40/54 (74%)	500	26	27208	31918	25	325	480	31	4	22.79	5.31	3057	3910	10	320
ZKServer ZKClient	4/ 4 (100%) 11/13 (84%)	4/ 4 (100%) 43/54 (79%)	1000	47	63524	76090	44	648	937	31	4	23.01	5.07	5719	7201	16	643

or quality of the tests. Moreover, we can see from Table 1 that for some models such as the ZooKeeper model, there are very large numbers of nodes and edges in both the SG and PG for, e. g., 1000 test cases executed. To deal with such large and complex models, we can use the user-defined search function discussed in Sect. 5.3 to limit the visualization to a subgraph. We do not show any subgraphs due to space limitations.

We use the Java server socket model to further discuss our experimental results based on the graphs obtained. The static visualization of the EFSM (see Fig. 9) shows the transition system and uses red edges to show expected excep-

tions, since the notion of failed tests does not apply. After applying abstractions, Fig. 10 shows the SG and PG for the Java server socket model with ten test cases executed, including failed transitions in red and labeled with (f).

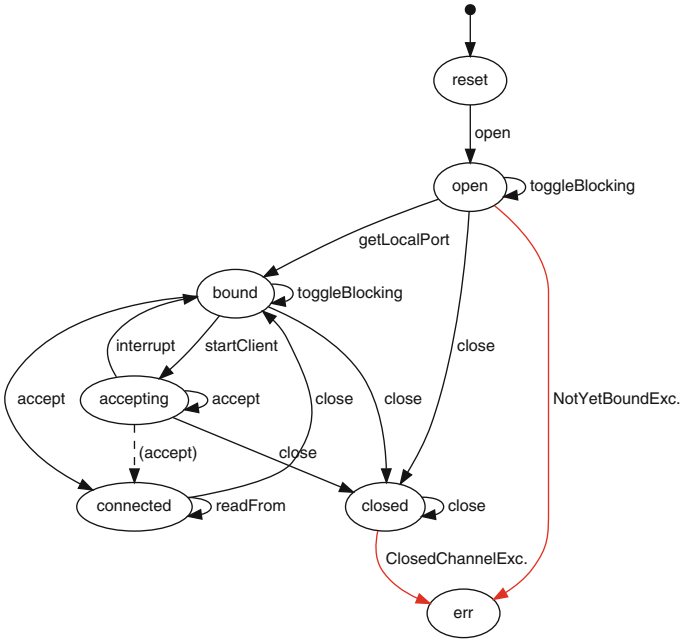


Fig. 9. EFSM for the Java server socket model.

Compared to the EFSM in Fig. 9, the SG in Fig. 10 shows the concrete executions instead of possible executions as shown by the EFSM. We see from the SG that all states have been visited after ten test cases; the SG also provides information about possible exceptions and failures occurred, actual paths and choices taken; the edge thickness indicates how often transitions were taken.

A good path-coverage-based testing strategy requires that the test cases execute as many linearly independent paths as possible. For the PG in Fig. 10, we can directly see that there are eight linearly independent paths. Each linearly independent path has a sequence of edges which represent executed transitions of the path. This gives us a simpler way of showing the paths as transition sequences, at the expense of a graph that has more nodes and edges overall. In addition, all loops, backtracked edges and taken choices are directly shown with their related linearly independent paths in the PG, and there is one linearly independent path which shows a failed test in the graph. Also, like the SG, the edge thickness in the PG indicates how often transitions were taken.

To show how our abstraction reduces the complexity of graphs, we use the Java server socket model as an example. Figure 11 shows the FSG without applying any abstractions for the Java server socket model with 10 test cases executed. This graph should be contrasted with the SG shown in Fig. 10(left). From this FSG, we notice that the FSG has many redundant edges between both state nodes and choice nodes, and it also has more choice nodes, as opposed to the SG

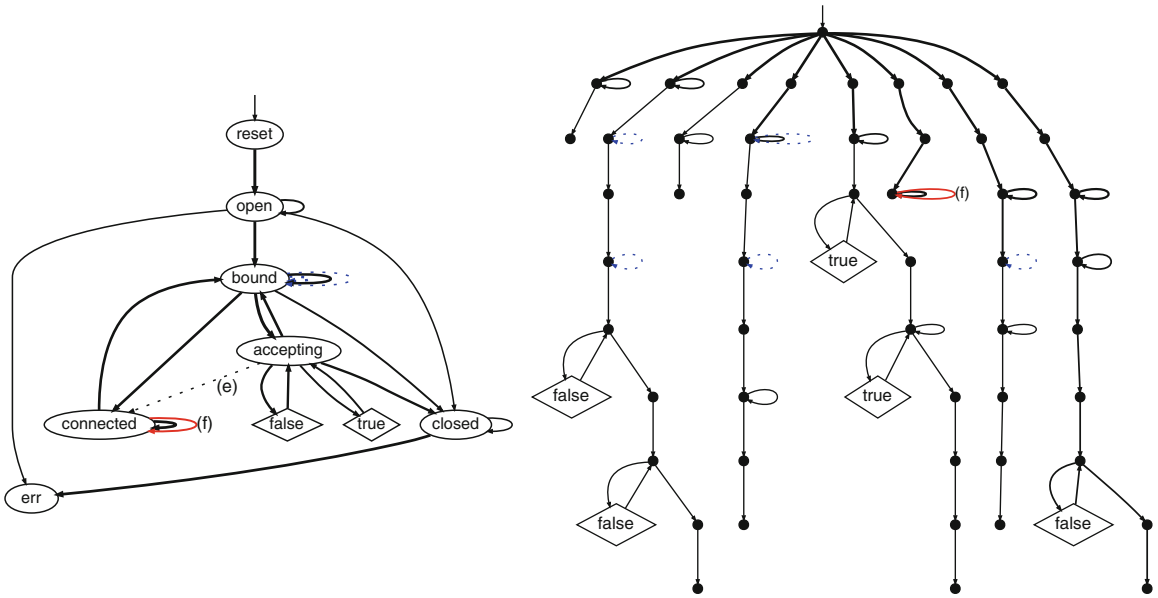


Fig. 10. SG (left) and PG (right) for the above model after ten tests.

in Fig. 10. Here, we do not show the FPG for the Java server socket model due to space limitations, but we give the detailed comparison between the SG and FSG and between the PG and FPG for the Java server socket model in Table 2. For instance, with 1000 test cases, the PG has three times fewer edges than the FPG; the SG has only 11 nodes and 27 edges, as compared to 61 nodes and 5491 edges in the FSG. This comparison shows that with the help of abstractions, the SG and PG are much more concise and less complex than the FSG and FPG.

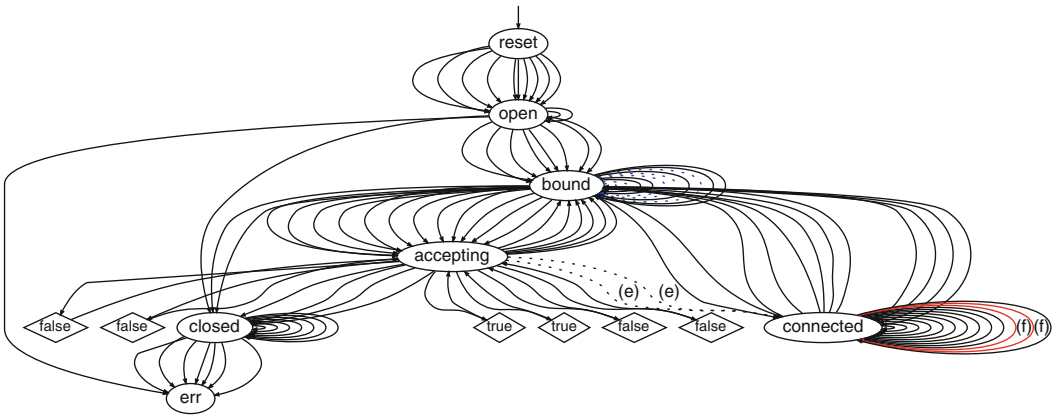


Fig. 11. FSG for the Java server socket model after ten tests.

7 Related Work

Coverage analysis is an important concern in software testing. It can be used as an exit criterion for testing and deciding whether additional test cases are

Table 2. Comparison between PG and FPG, and between SG and FSG of the Java server socket model.

Model	TC	PG				FPG				SG				FSG			
		Nodes	Edges			Nodes	Edges			Nodes	Edges			Nodes	Edges		
			E	FE	L		E	FE	L		E	FE	L		E	FE	L
JavaNio Server-Socket	10	57	79	1	17	70	97	2	21	9	23	1	6	13	109	2	31
	100	177	243	1	48	376	497	3	100	9	23	1	6	15	545	3	145
	200	363	528	4	111	811	1101	8	221	10	25	1	7	26	1239	8	353
	500	779	1147	8	247	1943	2613	14	520	11	27	1	8	40	2910	14	816
	1000	1269	1904	15	439	3721	4982	28	996	11	27	1	8	61	5491	28	1510

needed, and related to which aspects of the SUT. For source code coverage, tools generally only report a verdict on which line of code has been executed how often. In the tool Tecrevis, a visual representation of redundancy in unit tests provides a graphical mapping between each test case and the artifacts in the SUT (here: methods) that indicates which tests exercise the same component [13]. In path coverage, the underlying graph is usually derived from the source code, the *control flow graph*, or from the *call graph* of the SUT when considering function calls. In our approach, we are not directly concerned with visualizing paths of the SUT, but rather, paths on the testing model used for test-case generation. Correspondingly, our graphs are usually more concise than the control flow graph, as not all branches of the SUT may need to be modeled at the level of ESFMs. In particular, with respect to related work and the coverage analysis domain, visualization is usually an orthogonal concern to quantifying coverage, and not often considered.

Visualization makes coverage information understandable. Ladenberger and Leuschel address the problem of visualizing large state spaces in the ProB tool [14]. They introduce *projection diagrams,* which through a user-selectable function partition the states into equivalence classes. A coloring scheme for states and transitions indicates whether the state space has been exhausted, or all collapsed transitions share the same enablement. As their diagrams are based on the actually explored state space, they do not directly visualize coverage of the underlying model as in our approach. Moreover, they do not cover multiple transitions between the same pair of states as in our application scenario; however, this could be accounted for by adjusting the thickness of edges by the number of collapsed edges. Similarly, Groote and van Ham [10] applied an automated visualization to examples from the Very Large Transition System (VLTS) Benchmark set [8]. A relation between the graphical representation of the underlying model (in the form of UML sequence diagrams) and a set of paths from test cases is presented by Rountev et al. [19]. Their goal is deriving test cases, and as such they are not concerned with a representation of the paths.

The basic visualization elements of both SG and PG we have defined in this paper are based on the concept of *simple path* proposed by Ammann and Offutt [1]. An execution path is a simple path if there are no cycles in this path,

with the exception that the first and last states may be identical (the entire path itself is a cycle) [1]. Based on this definition, any execution path can be composed of simple paths. Therefore, in this paper, the concept of the simple path is applied by considering only transitions from $s_{origin}(t)$ to $s_{dest}(t)$ (or $s_{origin}(t)$ if t is a self-transition or backtracked transition).

8 Conclusions and Future Work

The main contribution of this paper is to present an approach to capture and visualize test-case-execution paths through models. This is achieved by first recording execution paths with a trie data structure, then visualizing them using state-based graphs (SGs) and path-based graphs (PGs) obtained by applying abstractions. The SG conveys the behavior of the model well. The PG only shows executed paths, without providing detail. It avoids crossing edges, which makes the PG more scalable, even though it contains more nodes and edges as such. Also, the PG directly indicates the number of linearly independent paths.

To obtain the SGs and PGs, we have proposed abstractions as our initial technique to reduce the size and complexity of graphs. We have implemented our approach as a path coverage visualizer for the Modbat model-based API tester. An experimental evaluation on several model-based test suites shows that our abstraction technique reduces the complexity of graphs, and our visualization of execution paths helps to show the frequency of transitions taken by the executed paths and to distinguish successful from failed test cases.

Future work includes investigating other techniques and tools to support more visualization features in the SGs and PGs, using more abstractions for the further reduction of larger graphs and applying the SGs and PGs not only for visualizing execution paths of models, but also for the SUT. Another direction of future work is to investigate approaches to perform state space exploration efficiently for selecting good test suites and visualizing execution paths. Although our current visualization approach has been applied to the Modbat tester, it is also possible to use it for other testing platforms. Furthermore, additional coverage metrics such as branch-coverage of boolean subexpressions within preconditions and assertions, or the more detailed *modified condition/decision coverage* (MC/DC) [7] could be used to refine the intermediate execution steps even further. In essence, many of the coverage techniques available at the SUT-level could be lifted to the model level.

References

1. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press, Cambridge (2016)
2. Artho, C.V., et al.: Modbat: a model-based API tester for event-driven systems. In: Bertacco, V., Legay, A. (eds.) HVC 2013. LNCS, vol. 8244, pp. 112–128. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03077-7_8

3. Artho, C., Rousset, G., Gros, Q.: Precondition coverage in software testing. In: Proceedings of 1st International Workshop on Validating Software Tests (VST 2016), Osaka, Japan. IEEE (2016)
4. AT&T Labs Research. Graphviz - Graph Visualization Software. https://www.graphviz.org
5. Brass, P.: Advanced Data Structures. Cambridge University Press, Cambridge (2008)
6. Cheng, K., Krishnakumar, A.: Automatic functional test generation using the extended finite state machine model. In: Proceedings of 30th International Design Automation Conference, DAC, pp. 86–91, Dallas, USA. ACM (1993)
7. Chilenski, J.J., Miller, S.P.: Applicability of modified condition/decision coverage to software testing. Softw. Eng. J. **9**(5), 193–200 (1994)
8. CWI and INRIA. The VLTS benchmark suite (2019). https://cadp.inria.fr/resources/vlts/. Accessed 20 May 2019
9. Gansner, E., Koutsofios, E., North, S.: Drawing graphs with dot (2006). http://www.graphviz.org/pdf/dotguide.pdf
10. Groote, J.F., van Ham, F.: Interactive visualization of large state spaces. Int. J. Softw. Tools Technol. Transf. **8**(1), 77–91 (2006)
11. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: Zookeeper: wait-free coordination for internet-scale systems. In: Barham, P., Roscoe, T. (eds.) 2010 USENIX Annual Technical Conference. USENIX Association (2010)
12. Jorgensen, P.C.: Software Testing: A Craftsman's Approach. Auerbach Publications, Boca Raton (2013)
13. Koochakzadeh, N., Garousi, V.: TeCReVis: a tool for test coverage and test redundancy visualization. In: Bottaci, L., Fraser, G. (eds.) TAIC PART 2010. LNCS, vol. 6303, pp. 129–136. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15585-7_12
14. Ladenberger, L., Leuschel, M.: Mastering the visualization of larger state spaces with projection diagrams. In: Butler, M., Conchon, S., Zaïdi, F. (eds.) ICFEM 2015. LNCS, vol. 9407, pp. 153–169. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25423-4_10
15. Lawrence, J., Clarke, S., Burnett, M., Rothermel, G.: How well do professional developers test with code coverage visualizations? An empirical study. In: IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 53–60. IEEE (2005)
16. Lu, S., Zhou, P., Liu, W., Zhou, Y., Torrellas, J.: Pathexpander: Architectural support for increasing the path coverage of dynamic bug detection. In: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 38–52. IEEE Computer Society (2006)
17. Myers, G.J., Badgett, T., Thomas, T.M., Sandler, C.: The Art of Software Testing, vol. 2. Wiley Online Library, Hoboken (2004)
18. Programming Methods Laboratory of École Polytechnique Fédérale de Lausanne. The Scala Programming Language. https://www.scala-lang.org
19. Rountev, A., Kagan, S., Sawin, J.: Coverage criteria for testing of object interactions in sequence diagrams. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, pp. 289–304. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31984-9_22
20. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann Publishers Inc., San Francisco (2007)

21. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Softw. Test. Verif. Reliab. **22**, 297–312 (2012)
22. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng. J. **10**(2), 203–232 (2003)

PAPER E

MULTI-OBJECTIVE SEARCH FOR MODEL-BASED TESTING

R. Wang, C. Artho, L. M. Kristensen, and V. Stolz

The 20th IEEE International Conference on Software Quality, Reliability, and Security, Vilnius, Lithuania, IEEE, 2020. (submitted)

Multi-objective Search for Model-based Testing

Rui Wang
Department of Computer Science, Electrical Engineering and Mathematical Sciences
Western Norway University of Applied Sciences
Bergen, Norway
rwa@hvl.no

Cyrille Artho
School of Electrical Engineering and Computer Science
KTH Royal Institute of Technology
Stockholm, Sweden
artho@kth.se

Lars Michael Kristensen
Department of Computer Science, Electrical Engineering and Mathematical Sciences
Western Norway University of Applied Sciences
Bergen, Norway
lmkr@hvl.no

Volker Stolz
Department of Computer Science, Electrical Engineering and Mathematical Sciences
Western Norway University of Applied Sciences
Bergen, Norway
vsto@hvl.no

***Abstract*—This paper presents a search-based approach relying on multi-objective reinforcement learning and optimization for test case generation in model-based software testing. Our approach considers test case generation as an exploration versus exploitation dilemma, and we address this dilemma by implementing a particular strategy of multi-objective multi-armed bandits with multiple rewards. After optimizing our strategy using the jMetal multi-objective optimization framework, the resulting parameter setting is then used by an extended version of the Modbat tool for model-based testing. We experimentally evaluate our search-based approach on a collection of examples, such as the ZooKeeper distributed service and PostgreSQL database system, by comparing it to the use of random search for test case generation. Our results show that test cases generated using our search-based approach can obtain more predictable and better state/transition coverage, find failures earlier, and provide improved path coverage.**

***Index Terms*—model-based testing, test case generation, bandit-based methods, multi-objective optimization, genetic algorithm, search-based software testing**

I. Introduction

The complexity of software systems today has amplified the importance of software testing as a scalable and efficient technique to discover defects. However, producing test cases by hand is tedious, expensive, and error-prone. *Model-based testing (MBT)* [1] addresses this problem by automatically generating test cases from abstract (formal) models of the *system under test (SUT)*. These abstract models encode the intended behaviors of the SUT and are often easier to develop and maintain compared to low-level test scripts [2]. However, for complex software systems, it is infeasible to explore and generate all the possible test cases for the software system under test. This means that a challenging decision needs to be made on how many tests cases to generate.

MBT is conducted via the automatic generation and execution of a *test suite*, that is, a set of *test cases*. Before starting MBT process, it is necessary to choose *test adequacy criteria* in order to evaluate the extent to which a test suite contains sufficient test cases. These criteria may consider the discovery of defects and obtaining a good code coverage. In this paper, our test adequacy criteria include state- and transition coverage, linearly independent path coverage, and the number of test cases needed to find the first failure.

It is a challenge for MBT to obtain test adequacy by generating a small test suite having few redundant test cases. Uncontrolled random approaches might result in test suites having redundant test cases which only cover few execution paths of the model and the SUT. Also, the decisions required to select a possible test case to generate faces the exploration versus exploitation dilemma when searching/exploring the state space. This dilemma can be described as finding a balance between: a) the exploration of diverse states/transitions which have not been selected to construct a test case; or have been selected fewer times, but might result in better addressing the test adequacy criteria; and b) the continuous exploitation of the states/transitions which have empirically resulted in better outcomes, with regard to the test adequacy criteria, when constructing a test case.

In this paper, we focus on the test case generation with a search-based approach relying on multi-objective reinforcement learning and optimization. Our aim is to find and generate a subset of test cases that optimizes the results when considering the chosen test adequacy criteria. We consider that 1) the process of test case generation is a problem that faces the exploration versus exploitation dilemma when searching/exploring possible test cases; 2) obtaining good and balanced results of the chosen test adequacy criteria with fewer generated test cases is a multi-objective optimization problem. In particular, we want to implement an efficient search that (1) does not require user-defined weights, which rely on domain knowledge; and (2) adjusts the choices dynamically based on the coverage of previous tests.

The main contribution of this paper is to present a search-based test case generation approach combining: 1) generation

of test cases based on a particular strategy of multi-objective multi-armed bandits with multiple rewards; and 2) optimizing the chosen adequacy criteria with a Pareto-efficient multi-objective genetic algorithm, in the form of the non-dominating sorting genetic algorithm (NSGA-II) [3]. We evaluate our approach on several models developed for the Modbat model-based API tester [4] by comparing our search-based testing with the random testing. Our experiments show that our search-based approach can obtain more predictable and better results of the chosen adequacy criteria compared to random test case generation, when considering the trade-offs of the criteria.

A second contribution is an implementation of our bandit-based search strategy in the Modbat model-based API tester, which is a new feature for the Modbat 3.4 release [5]. We define test adequacy criteria as multi-objectives so that Modbat implements our search-based test case generation in addition to its standard random search. The test adequacy criteria are optimized using the jMetal [6], [7] multi-objective optimization framework which applies the NSGA-II algorithm on the Modbat models that we use as training set to find a Pareto optimal solution set having reward parameter settings. The parameter settings in the Pareto optimal solution set is then used as inputs to our bandit-based search strategy to generate test cases for other advanced Modbat models and targeting the chosen test adequacy criteria.

The rest of this paper is organized as follows. Section II provides background on the Modbat model-based API tester, the definition of execution paths as test cases, and the test adequacy criteria. In Section III, we present our approach of multi-objective test case generation and optimization. Section IV presents our experimental evaluation and analyzes the results obtained from the experiments. Section V discusses related work, and in Section VI, we conclude and discuss future work.

II. Model-based Software Testing

Our work assumes that a mechanism for automated test execution of a system under test (SUT) is provided in the form of a test harness and properties (such as assertions) about the behavior of the SUT. In addition to executing test cases automatically, MBT can also generate inputs (or calls) to the SUT automatically, and verify that its output matches the expected output [1]. We introduce MBT of state-based systems in the context of the Modbat tester [4].

A. Modbat Model-based API Tester

Modbat is a model-based testing tool that performs online testing of state-based systems that runs on a Java Virtual Machine [4]. Modbat uses extended finite state machines (EFSMs) [8] as its theoretical foundation and implements extensions in a domain-specific language based on Scala [9]. The EFSMs used by Modbat is formally defined as:

Definition 1 (Extended Finite State Machine [10])**.** *An extended finite state machine is a tuple* $M = (S, s_0, V, A, T)$ *such that:*

- S *is a finite set of states, including an initial state* s_0.
- $V = V_1 \times \ldots \times V_n$ *is an* n*-dimensional vector space representing the set of values for variables.*
- A *is a finite set of actions* $A : V \rightarrow (V, R)$*, where* $res \in R$ *denotes the* result *of an action, which is either* successful, failed, backtracked, *or* exceptional. *A successful action allows a test case to continue; a failed action constitutes a test failure and terminates the current test; a backtracked action corresponds to the case where the enabling function of a transition is* false *[8]; exceptional results are defined as such by user-defined predicates that are evaluated at run-time, and cover the non-deterministic behavior of the SUT. We denote by* $Exc \subset R$ *the set of all possible exceptional outcomes.*
- T *is a transition relation* $T : S \times A \times S \times E$*; for a transition* $t \in T$*, we denote the left-side (origin) state by* $s_{origin}(t)$ *and the right-side (destination) state by* $s_{dest}(t)$*, and use the shorthand* $s_{origin} \rightarrow s_{dest}$ *if the action is uniquely defined. A transition includes a possible empty mapping* $E : Exc \rightarrow S$*, which maps exceptional results to a new destination state.*

Listing 1 illustrates a Modbat model of a garage door control system that we will use as a running example to introduce the basic concepts of Modbat. A valid execution path in a Modbat model starts from the initial state (automatically derived from the first declared state) and consists of a sequence of transitions. Transitions are declared with a concise syntax: "$origin$" $\rightarrow$ "$dest$" := $\{action\}$. The GarageDoorTester model in Listing 1 consists of five states: "$DoorUp$", "$DoorClosing$", "$DoorDown$", "$DoorOpening$", and "End". The initial state is "$DoorDown$" in Line 4.

The GarageDoorTester model tests the garage door system shown in Listing 2 (only fields, public methods and the ***stop*** private method are shown due to page limitations). The garage door system controls the opening and closing of a 2-meter garage door using ***open*** and ***close*** methods to set a door motor with a speed $+0.125m/s$ or $-0.125m/s$, respectively (Line 14 and Line 21 in Listing 2). The system uses a private method ***waitLimitHit*** (Line 18 and Line 25) to check the status of the door every second and it calls the ***stop*** private method (Line 28) when the door is fully open or closed. The system takes 16 seconds to open or close the garage door, as implemented by ***waitLimitHit*** method. When the garage door is fully open or closed, the speed of the door motor is set to zero and the motor is stopped by the ***stop*** private method.

Modbat has built-in *require* and *assert* methods. The GarageDoorTester model in Listing 1 uses the *require* method (Line 5, Line 6, Line 14 and Line 15) in transitions to check if preconditions are fulfilled. Preconditions must be fulfilled in order for a transition to be enabled. For example, if the *require* methods in Line 5 and Line 6 expressing preconditions are satisfied, then the transition "$DoorDown$" $\rightarrow$ "$DoorOpening$" is enabled. The ***open*** method is then called to open the garage door. The preconditions are similar for the transition "$DoorUp$" $\rightarrow$ "$DoorClosing$" that calls the ***close*** method. The attribute ***stay*** of a transition is used to delay (in this case

16 seconds) while waiting for the door to be fully open or closed. After the ***open*** or ***close*** method is called and the corresponding transition is executed, the *assert* methods in Line 10 and Line 11 in transition "*DoorOpening*" → "*DoorUp*", or in Line 19 and Line 20 in transition "*DoorClosing*" → "*DoorDown*", are used as assertions to check that the status of the door and door motor is correct when the door is fully open or closed.

```
1  class GarageDoorTester extends Model {
2    val garage = new GarageDoor()
3    // transitions
4    "DoorDown" -> "DoorOpening" := {
5      require(gargage.doorFullyClosed)
6      require(garage.motorStopped)
7      garage.open()
8      } stay 16000
9    "DoorOpening" -> "DoorUp" := {
10     assert (garage.doorFullyOpen)
11     assert (garage.motorStopped)
12   }
13   "DoorUp" -> "DoorClosing" := {
14     require(gargage.doorFullyOpen)
15     require(garage.motorStopped)
16     garage.close()
17   } stay 16000
18   "DoorClosing" -> "DoorDown" := {
19     assert (garage.doorFullyClosed)
20     assert (garage.motorStopped)
21   }
22   "DoorDown" -> "End" := skip
23   "DoorUp" -> "End" := skip
24 }
```

Listing 1: Modbat model GarageDoorTester.

Modbat actions (which execute code related to transitions) have four possible outcomes: successful, backtracked, failed, or exceptional. Given the different possible outcomes of Modbat actions, different rewards of our bandit-based search strategy are defined in Section III. A *successful* action allows a test case to continue with another transition, if available. An action is *backtracked* and resets the transition to its original state if any of its preconditions are violated. An action *fails* if an assertion is violated, if an unexpected exception occurs, or if an expected exception does not occur. In our GarageDoorTester example, the action of transition "*DoorUp*" → "*DoorClosing*" is backtracked if any *require* methods in the action evaluate to false, and the action fails if any *assert* methods evaluate to false in, e. g., "*DoorClosing*" → "*DoorDown*". If no preconditions or assertions are violated, and no exceptional result occurs, the action is *successful*.

B. Execution Paths and Test Cases

For Modbat models, a finite *execution path* consists of a sequence of transitions starting from the initial state and leading to a *terminal state* (a state without outgoing transitions, or a state after a test failed). Each finite execution path represents a test case generated from a Modbat model. That is, a test case is an execution path consisting of a sequence of transitions. Execution paths of Modbat models are formally defined as:

```
1  class GarageDoor {
2    val garageTopHeight = 2d       // two meters
3    val garageBottomHeight = 0d
4    val motorSpeeds = Map[String, Double]("Zero"
       -> 0.0, "PlusSpeed" -> 0.125, "MinusSpeed"
       -> -0.125)
5    // initial door close
6    var currentDoorHeight = garageBottomHeight
7    // initial motor speed 0.0
8    var motorSpeed = motorSpeeds("Zero")
9    var motorStopped = true
10   var motorUp = false
11   var motorDown = false
12   var doorFullyOpen = false
13   var doorFullyClosed = true
14   def open() {
15     motorUp = setMotorSpeed("PlusSpeed")
16     if(motorUp) {
17         doorFullyClosed = false
18         waitLimitHit()
19     }
20   }
21   def close() {
22     motorDown = setMotorSpeed("MinusSpeed")
23     if(motorDown) {
24         doorFullyOpen = false
25         waitLimitHit()
26     }
27   }
28   private def stop() {
29     motorStopped = setMotorSpeed("Zero")
30     if (motorStopped){
31         currentDoorHeight match {
32             case garageTopHeight
33                 => doorFullyOpen = true
34             case garageBottomHeight
35                 => doorFullyClosed = true
36         }
37     }
38   }
39   ...
40 }
```

Listing 2: Garage door system.

Definition 2 (Execution Path [10])**.** *Let* $M = (S, s_0, V, A, T)$ *be an EFSM. A* finite execution path p *of* M *is a sequence of transitions, which constitute a path* $p = t_0 t_1 \dots t_n, \quad t_n \in T$, *such that* $s_{origin}(t_0) = s_0$, *the origin and destination states are linked:* $\forall i, 0 < i \leq n, s_{origin}(t_i) = s_{dest}(t_{i-1})$, *and* $s_{dest}(t_n) \in S_{terminal}$; $S_{terminal}$ *is the set of terminal states.*

C. Test Adequacy Criteria as Multi Objectives

For MBT, test adequacy criteria are often chosen to guide the automatic test case generation so that it produces a good test suite [1]. Modbat supports test adequacy criteria including state- and transition coverage [4], and linearly independent path coverage [10]. The state- and transition coverage indicates the number of states and transitions, respectively, that have been explored by a test suite. A linearly independent path (LIP) is any path through a program that contains at least one new path edge (transition) which is not included in any other linearly independent path [10]. Therefore, the linearly independent path coverage indicates the execution paths covered by a test suite.

These test coverage metrics can be measured as the outcome of the executed test suite and visualized (along with the test model) using Graphviz [4], [10] by Modbat. In addition to coverage, Modbat can also provide the measurement of failures found after a test suite is executed [4]. Thus, for our test case generation approach, we choose four different test adequate criteria: 1) state coverage (Cov_s); 2) transition coverage (Cov_t); 3) linearly independent path (LIP) coverage (Cov_{lip}); and 4) the number of test cases used to find the first failure ($NTest_{fail_t}$). We use these test adequacy criteria as objectives for multi-objective optimization.

Prior to this work, Modbat already supported static weights for transition choices, which affect the likelihood of choosing a given transition. However, a good setting of these weights requires insight into the semantics of the model, and the weights remain fixed during the entire test generation process.

III. Multi-objective Optimization

A test suite consists of a set of test cases derived from the test model, and each test case represents one execution path which in turn consists of a sequence of transitions. The generation of a test case therefore relies on the decisions made in each step to select the transitions that are to be part of the constructed execution paths. As introduced earlier, the decision made to select a transition faces the exploration versus exploitation dilemma in terms of finding a balance between: a) the exploration of different transitions which have not yet been selected; or have selected fewer times; and b) the continuous exploitation of already selected transitions which have empirically resulted in better outcomes (e. g., a high coverage). Reinforcement learning [11] is the subfield of machine learning devoted to studying problems and designing algorithms that analyze this dilemma. The multi-armed bandit problem, extensively studied by Berry and Fristedt [12], is a well-established class of sequential decision problems in the context of reinforcement learning.

A. Bandit Search-based Test Case Generation

Bandit problems consider a player (agent) that needs to choose among K arms (actions) in I rounds on a multi-armed bandit slot gambling machine. The objective is to maximize the cumulative reward (money) as much as possible in a casino by consistently taking the optimal arm (action) over rounds [13]. At each round $i = 1, \ldots, I$, the player selects an arm (action) $j \in \{1, \ldots, K\}$ and receives the reward $r_{(j,i)}$ (money). The player (agent) has a goal: on one hand, finding out (exploit) which arm could be currently optimal to have the highest expected reward; on the other hand, exploring other arms (actions) that currently are not optimal, but may turn out to be optimal in the long run [13], [14], [15].

Several algorithms, such as ϵ-greedy [11], Boltzmann Exploration (Softmax) [16], and Reinforcement Comparison [11] have been proposed to solve bandit problems. In our approach, we rely on the Upper Confidence Bounds (UCB) family [15] of algorithms. For reinforcement learning, the regret is one popular measure of a policy's success in addressing the exploration versus exploitation dilemma. The regret is the expected loss due to the fact that the policy does not always play the best (optimal) action [15]. Compared to other algorithms, the UCB family has been theoretically analyzed and has an expected optimal logarithmic growth of regret uniformly over time [15], [13]. An extension of UCB-style algorithms has proven very successful in computer Go [17]. Lai and Robbins [18] showed that the regret for the multi-armed bandit problem has to grow at least logarithmically in the number of rounds. We use the UCB1 bandit algorithm from the UCB family [15] as a basis for implementing our multi-objective search strategy for test case generation.

The UCB1 algorithm operates as follows:

a) each bandit arm is played once at the initialization of the algorithm.

b) afterwards, the algorithm iteratively plays bandit arm j that maximizes

$$\bar{x}_j + \sqrt{\frac{2 \ln n}{n_j}} \tag{1}$$

where $\bar{x}_j$ is the average reward (in $[0, 1]$) from arm j, n_j is the number of times arm j was played, and n is the overall number of plays so far.

The UCB1 algorithm indicates that the reward term $\bar{x}_j$ encourages the exploitation of higher reward arms, while the term $\sqrt{\frac{2 \ln n}{n_j}}$ encourages the exploration of less-visited arms [15].

Based on the UCB1 algorithm, we consider each transition $t_j \in T$ to select for constructing an execution path (a test case) as a bandit arm to play. We denote the reward function as $r : T \to \mathbb{R}$. After executing a transition $t_j \in T$, its corresponding immediate reward $r_{t_j} \in \mathbb{R}$ is received accumulatively, and computed as $r_{t_j} = \bar{r}_{t_j} + \hat{r}_{t_j}$, where $\bar{r}_{t_j}$ is a transition outcome average reward iteratively accumulated, and $\hat{r}_{t_j}$ is a transition action expected reward iteratively accumulated. All rewards are in the interval $[0, 1]$, and we show how to compute them shortly.

The above implies that our bandit heuristic search (BHS) strategy for test case generation becomes the following:

a) each transition $t \in T$ is selected once at the initialization of the strategy.

b) afterwards, the strategy iteratively select a transition $t_j \in T$ that maximizes

$$\bar{r}_{t_j} + \hat{r}_{t_j} + \sqrt{\frac{2 \ln n_{s_{origin}(t_j)}}{n_{t_j}}} \tag{2}$$

where $\bar{r}_{t_j}$ is the transition outcome average reward (in $[0, 1]$) for transition t_j, $\hat{r}_{t_j}$ is the transition action expected reward for transition t_j, n_{t_j} is the number of times transition t_j was selected, and $n_{s_{origin}(t_j)}$ is the number of times that the origin state s_{origin} of the transition t_j is visited and used to select transitions.

This strategy indicates that the reward terms $\bar{r}_{t_j}$ and $\hat{r}_{t_j}$ jointly encourage the exploitation of higher rewarded transitions, while the term $\sqrt{\frac{2 \ln n_{s_{origin}(t_j)}}{n_{t_j}}}$ encourages the exploration of less-selected transitions.

To iteratively compute a transition outcome average reward $\bar{r}_{t_j}$, we consider four types of transition outcome rewards as a set of rewards $\mathbb{R}_{to}$ including the r_{to_self}, $r_{to\_success}$, r_{to_back}, and r_{to_fail}. Among these four types, the r_{to_self} is a self-transition reward for a successful transition that has $s_{origin} = s_{dest}$, while the $r_{to\_success}$ is a reward given to a successful transition that has $s_{origin} \neq s_{dest}$. The r_{to_back} is the reward for a backtracked transition, and the r_{to_fail} is the reward for a failed transition. We denote a transition outcome reward received at the i'th iteration for a transition $t_j \in T$ by $r_{to_{(t_j,i)}} \in \mathbb{R}_{to}$, and n_{t_j} denotes the number of times transition t_j was selected. Therefore, we compute the accumulated transition outcome average reward $\bar{r}_{t_j}$ for a transition t_j using Equation 3:

$$\bar{r}_{t_j} = \frac{1}{n_{t_j}} \sum_{i=1}^{n_{t_j}} r_{to_{(t_j,i)}} \tag{3}$$

The excepted transition action reward $\hat{r}_{t_j}$ is the sum of the given rewards for pass/fail, weighted by how many times the two verdicts actually occurred. To compute an expected reward iteratively, we take into account four different rewards for two types of transition actions (precondition and assertion) as a set of rewards $\mathbb{R}_{ta}$ including the passed precondition reward $r_{precond\_pass}$, failed precondition reward $r_{precond\_fail}$, passed assertion reward r_{assert_pass}, and failed assertion reward r_{assert_fail}. Then, the excepted transition action reward $\hat{r}_{t_j}$ for a transition t_j can be computed using Equations 4, 5 and 6.

$$\hat{r}_{t_j} = \hat{r}_{t_j\_precond} + \hat{r}_{t_j\_assert} \tag{4}$$

$$\begin{aligned} \hat{r}_{t_j\_precond} = &\frac{C_{precond\_pass}}{C_{precond\_total}} \times r_{precond\_pass} \\ &+ \frac{C_{precond\_fail}}{C_{precond\_total}} \times r_{precond\_fail} \end{aligned} \tag{5}$$

$$\begin{aligned} \hat{r}_{t_j\_assert} = &\frac{C_{assert\_pass}}{C_{assert\_total}} \times r_{assert\_pass} \\ &+ \frac{C_{assert\_fail}}{C_{assert\_total}} \times r_{assert\_fail} \end{aligned} \tag{6}$$

In Equation 4, $\hat{r}_{t_j\_precond}$ represents the expected precondition (action) reward for the transition t_j; $\hat{r}_{t_j\_assert}$ represents the expected assertion (action) reward for the transition t_j. Likewise, the *counts* for passed and failed preconditions and assertions used in Equations 5 and 6, as well as their total number, are updated during each iteration. The overall steps for test case generation with our bandit heuristic search strategy are summarized in pseudocode in Algorithm 1. This is the heuristic search strategy that we have implemented in Modbat. We explain the steps related to our search strategy, without showing the pseudocode for how transitions and test cases are executed. Modbat initializes a list of transitions *transitions* and s from an initial state s_0. We need to initialize the number of test cases n, all counter variables (with 0 values), and reward variables. In Line 4, the function EXECUTETRANSITIONS generates and executes a test case consisting of a sequence of selected transitions from an initial state s_0 to a terminal state $s_{terminal}$. In Line 6 in function EXECUTETRANSITIONS, the function BANDITHEURISTICSEARCH is invoked to select a transition *trans* using our bandit heuristic search strategy. Then, this selected transition *trans* is executed by the function EXECUTETRANSITION shown in Line 7, with a transition result of type *result* as the function return value. Meanwhile, function EXECUTETRANSITION calls the function UPDATEEXPECTEDREWARD in Line 16 to update the transition action expected reward $\hat{r}_{t_j}$ for the selected transition *trans* based on Equations 4, 5 and 6. After receiving the return value *result* in Line 7, the function UPDATEAVERAGEREWARD in Line 30 updates the transition outcome average reward $\bar{r}_{t_j}$ for *trans* with Equation 3, based on the result type of *trans*.

B. Bandit Search-Based Test Suite Optimization with JMetal

The aim of our bandit heuristic search strategy is to guide the test case generation with the objective of addressing the test adequacy criteria with smaller test suites containing less redundant test cases. The strategy relies on the configuration of eight different rewards to initialize the test case generation (as shown by the ***Require*** in Algorithm 1). Therefore, to achieve our aim, we need to find optimal solutions to configure these rewards and obtain optimized test adequacy criteria (objectives) defined as in Section II-C, while considering the trade-offs of these criteria. Thus, we consider our bandit heuristic search strategy as a multi-objective optimization problem: tune our bandit heuristic search strategy shown in Algorithm 1 to find the optimal solutions. With these optimal solutions found, we can use them for the test case generation of Modbat models in general with our strategy.

Formally, we assume for our multi-objective bandit search optimization problem that a solution can be described in terms of an 8-dimensional reward decision vector $\vec{r}$ in the reward decision space $\mathcal{R}^8$. Such a solution can be used to initialize the generation of a test suite $ts \in TS$ (initialization of Algorithm 1), where TS is a set of test suites. Then, the vector-valued objective function $\vec{f} : \mathcal{R}^8 \rightarrow \mathcal{O}$ evaluates the quality of a specific solution by assigning it an objective vector $\vec{o} = \vec{f}(\vec{r})$ in the objective space $\mathcal{O}$. We define the reward decision vector as

$$\begin{aligned} \vec{r} = &(r_{to\_self}, r_{to\_success}, r_{to\_back}, r_{to\_fail}, r_{precond\_pass}, \\ &r_{precond\_fail}, r_{assert\_pass}, r_{assert\_fail}), \end{aligned} \tag{7}$$

and the objective vector with our four objectives (test adequacy criteria) as

$$\begin{aligned} \vec{o} &= (f_1(\vec{r}), f_2(\vec{r}), f_3(\vec{r}), f_4(\vec{r})) \\ &= (Cov_s, Cov_t, Cov_{lip}, NTest_{fail1}), \end{aligned} \tag{8}$$

according to our test adequacy criteria defined in Section II-C. Without loss of generality, it is assumed that all objectives are all equally important and our goal is to optimize them.

Algorithm 1 Bandit Heuristic Search for Test Case Generation

Require: Initialize s, $transitions$, n, r_{to_self}, $r_{to\_success}$, r_{to_back}, r_{to_fail}, $r_{precond\_pass}$, $r_{precond\_fail}$, r_{assert_pass}, r_{assert_fail}, $C_{precond\_pass}$, $C_{precond\_fail}$, C_{assert_pass}, C_{assert_fail}.

function EXECUTETESTS
  for i = 1 to n **do** ▷ n:number of test cases
    EXECUTETRANSITIONS

function EXECUTETRANSITIONS
  while s is not a $s_{terminal}$ **do** ▷ s:current state, starting from s_0
    $trans \leftarrow$ BANDITHEURISTICSEARCH($transitions$)
    $result \leftarrow$ EXECUTETRANSITION($trans$)
    UPDATEAVERAGEREWARD($result$, $trans$)

function BANDITHEURISTICSEARCH($transitions$)
  if $transitions$ has any never selected transitions **then**
    return $t_{1st\_unselected}$ in $transitions$
  else
    return $t_{j'}$ in $transitions$ having argmax$\{\bar{r}_{t_j} + \hat{r}_{t_j} + \sqrt{\frac{2 \ln n_{s_{origin}(t_j)}}{n_{t_j}}}\}$

function EXECUTETRANSITION($trans$)
  UPDATEEXPECTEDREWARD($trans$)

function UPDATEEXPECTEDREWARD($trans$)
  if precondition of $trans$ **then**
    if pass **then**
      update $C_{precond\_pass}$ += 1
    else
      update $C_{precond\_fail}$ += 1
    update $C_{precond\_total} = C_{precond\_pass} + C_{precond\_fail}$
  if assertion of $trans$ **then**
    if pass **then**
      update C_{assert_pass} += 1
    else
      update C_{assert_fail} += 1
    update $C_{assert\_total} = C_{assert\_pass} + C_{assert\_fail}$
  update $\hat{r}_{trans} = \hat{r}_{t_{trans\_precond}} + \hat{r}_{t_{trans\_assert}}$ for $trans$ with Equations 5,6

function UPDATEAVERAGEREWARD($result$, $trans$)
  switch $result$ **do**
    case $success$
      $r_{to_{(trans,i)}} = r_{to\_success}$
    case $self$
      $r_{to_{(trans,i)}} = r_{to\_self}$
    case $backtracked$
      $r_{to_{(trans,i)}} = r_{to\_back}$
    case $failed$
      $r_{to_{(trans,i)}} = r_{to\_fail}$
  update $\bar{r}_{trans} = \frac{1}{n_{trans}} \sum_{i=1}^{n_{trans}} r_{to_{(trans,i)}}$ for $trans$

Therefore, to solve our multi-objective bandit search optimization problem, we need to find those reward decision vectors as solutions that optimize the vector-valued objective function $\vec{f} : \mathcal{R}^8 \rightarrow \mathcal{O}$. These solutions balance the trade-offs between the objectives, and we measure the optimality of the solutions through the concepts of *Pareto optimality* and *Pareto dominance* [19], [20], [21].

Following the concept of *Pareto dominance*, given two solutions $\vec{r} \in \mathcal{R}^8$ and $\vec{r}' \in \mathcal{R}^8$ as reward decision vectors which can be used to initialize two test suites ts and ts', $\vec{r}$ is said to dominate, or *Pareto-dominate*, $\vec{r}'$ (written as $\vec{r} \succ \vec{r}'$) if and only if their objective vectors $\vec{o} = \vec{f}(\vec{r})$ and $\vec{o}' = \vec{f}(\vec{r}')$ satisfy: $\forall i \in \{1, 2, \ldots, k\}, \vec{f}(\vec{r}) \geqslant \vec{f}(\vec{r}') \wedge \exists i \in \{1, 2, \ldots, k\} : \vec{f}(\vec{r}) > \vec{f}(\vec{r}')$. Here, we use $k = 4$ since we have four test adequacy criteria used as objectives. All reward decision vectors that are not dominated by any other reward decision vectors are said to form the *Pareto optimal set* $\mathcal{R}^{8*} \subseteq \mathcal{R}^8$, while the corresponding objective vectors are said to form the *Pareto frontier* $\mathcal{O}^* = \vec{f}(\mathcal{R}^{8*}) \subseteq \mathcal{O}$. That is, the *Pareto optimal set* $\mathcal{R}^{8*}$ contains only non-dominating reward decision vectors as optimal solutions to our multi-objective bandit search optimization problem. Each non-dominating reward decision vector can then be used to initialize the generation of a test suite. This means that we find an optimal subset of test suites $TS^* \subseteq TS$ which balance the trade-offs of our four different test adequacy criteria: no other subset of TS can improve one objective without making another objective worse.

To obtain a *Pareto optimal set* $\mathcal{R}^{8*}$ for our multi-objective bandit search problem, we apply the jMetal [6], [7] Java-based framework for multi-objective optimization using meta-heuristics. jMetal is specifically oriented towards multi–objective optimization, and implements a number of state-of-the-art multi–objective optimization algorithms, such as the NSGA-II [3]. NSGA-II is one of the most well-known and widely used multi-objective evolutionary algorithm to obtain the *Pareto optimal set*.

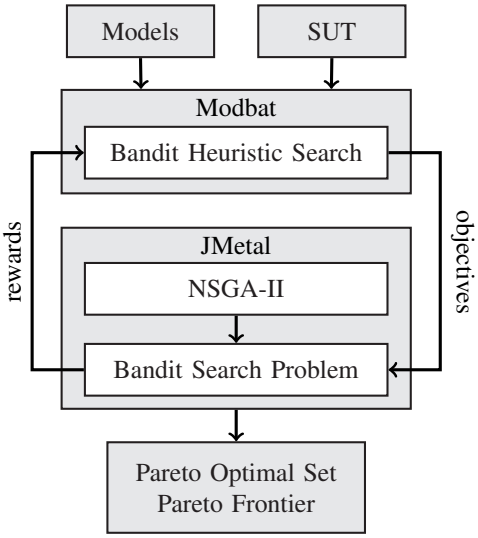


Fig. 1: Multi–objective bandit-search optimization.

Fig. 1 gives an overview of our implementation used to solve our multi–objective bandit search optimization problem for Modbat models with the aid of jMetal v5 [7] and NSGA-II. The working principle of jMetal is based on algorithms (such as NSGA-II) chosen by users and user-defined problems to solve. Users need to first define their multi–objective optimization problems with objective functions, and then solve them with the chosen algorithm. We have implemented our

multi–objective bandit search optimization problem in jMetal with our defined vector-valued objective function $\vec{f} : \mathcal{R}^8 \rightarrow \mathcal{O}$. Then we use the NSGA-II genetic algorithm provided by jMetal to solve this problem. The process goes through evaluation rounds of the NSGA-II algorithm with the number of rounds and population provided. For each evaluation round, we run different Modbat models in parallel using 8 different values of rewards (generated by the NSGA-II from jMetal) as the input parameters for our bandit heuristic search strategy. After the Modbat models are executed, the results of our defined four test adequacy criteria for all models are sent to jMetal as the values of the objectives which can then be used by the NSGA-II algorithm to generate reward values for the next evaluation round. When all evaluation rounds of the NSGA-II algorithm are finished, jMetal provides files containing the *Pareto optimal set* and the *Pareto frontier* found by the NSGA-II. The detailed configuration of our experiment will be discussed in Section IV.

IV. Experimental Evaluation

We have evaluated our bandit heuristic search strategy on a collection of Modbat models which have earlier been used to generate test cases with the standard random approach provided by Modbat.

A. Experimental Setup

The collection of models that we consider includes four simple models as a training set and two large and complex models as the test set. The simple models in the training set encompass the *ChooseTest* model [10], the *Java Server Socket* model [22], the *Java Array List* model [23], and the *Java Linked List* model [23]. The large complex models in the test set are the *ZooKeeper* [24] and *PostgreSQL* [25] models. The Java Array and Linked List models, PostgreSQL model, as well as the ZooKeeper model, consist of several parallel EFSMs, which are executed in an interleaved way [4]. Table I summarizes the total numbers of states, transitions, numbers of different EFSMs, and non-commenting lines of code (NCLOC) for each model. Note that states refer to labeled states, which in EFSMs are augmented with variables that may be from a potentially infinite-sized domain; therefore, the full number of *extended* model states is usually in thousands or millions per test. Moreover, in Table I, we summarize the *declared* states of all types of models involved, but we do not count states of multiple model instances in a given test multiple times. Likewise, transitions may include internal choices over different functions, or invoke functions that are arbitrarily complex. This means that the numbers in Table I only give a picture of the syntactic complexity of a model.

Specifically, we first apply our strategy on the four simple Modbat models in the training set using jMetal to optimize our strategy. Then, the weights of the eight rewards in the resulting Pareto optimal set are used in Modbat's configuration as the optimal parameters for our strategy on the two large complex Modbat models in the test set.

Table I: Number of declared states, transitions, EFSMs, and code size for each model for the evaluation

Model	States	Transitions	EFSM(s)	NCLOC
ChooseTest	3	3	1	10
JavaServerSocket	7	17	1	105
ArrayList	5	51	3	392
LinkedList	5	59	3	428
PostgreSQL	13	15	2	414
ZooKeeper	17	58	2	2225

The experiments have been performed using an Ubuntu 18.04.4 LTS (GNU/Linux 4.15.0-88-generic x86_64) on an Intel(R) Xeon(R) Gold 6136 CPU 3.00GHz (48 CPUs). For the experimental setup of Modbat, we configure that each Modbat model runs 40 test suites. We preconfigure the seed for the random number generator and fixed 40 seeds to make the test generation deterministic. Each test suite consists of 50 test cases.

To configure the NSGA-II algorithm provided in jMetal, we use its default settings except that the population size is set to 50, and the maximum number of generations in the evolutionary search is set to 100. The reason for using these two relatively small values is to keep the time used for the experimental evaluation manageable. Furthermore, for the optimization process, we configure Modbat and jMetal to run the four simple models in parallel. That is, for each evaluation, jMetal provide the values of 8 rewards as a parameter setting for Modbat to run four models in parallel (random values for 8 rewards as the initialization). The resulting values of the four test adequacy criteria of each model are then collected and used as objectives for jMetal to execute NSGA-II (16 objectives together in total due to 4 models). All collected resulting values of four test adequacy criteria are within 0 to 100, which are defined or computed as follows:

- State coverage: $Cov_s \in \{0, \ldots, 100\}$
- Transition coverage: $Cov_t \in \{0, \ldots, 100\}$
- Score of Cov_{lip}: $Cov_{lip} * 2$, $Cov_{lip} \in \{1, \ldots, 50\}$
- Score of $NTest_{fail1}$: $102 - NTest_{fail1} * 2$, $NTest_{fail1} \in \{1, \ldots, 51\}$

The two last scores are calculated based on the fact that with 50 tests, at most 50 linearly independent paths are possible, and that the best possible outcome is if the first test finds a failure; if no test finds a failure, we count the score as if the 51st test (which is never tried) would have found it.

As each parameter setting for 8 rewards was tested with 40 seeds and 50 test cases per seed on four models, we ran 8,000 tests per parameter setting to determine fitness. With a population size of 50 and 100 generations, we ran a total number of 40 million tests in the training phase, which took four days using a 48 CPUs cluster. For the ZooKeeper model, we just apply 50 optimal solution candidates in the Pareto optimal set directly and collect the results for the test adequacy criteria; while for the PostgreSQL, we perform mutation testing using 86 mutants to inject 86 different errors to the PostgreSQL. Then, we apply 50 optimal solution candidates to 86 mutated PostgreSQL, respectively.

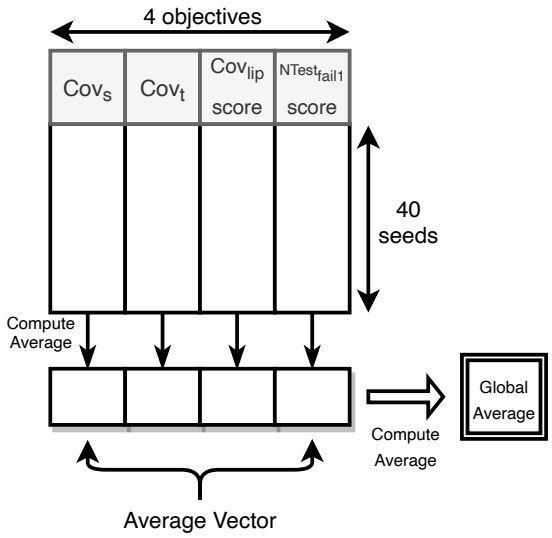


Fig. 2: Result computations for the random approach.

B. Data Post-processing

We have compared the performance of our bandit heuristic search strategy and its optimization to the random approach already provided by Modbat. Fig. 2 shows the basic process of collecting and computing result data of the random approach for each model. We compute the average of collected resulting values for each objective obtained using the 40 fixed seeds to get an average result vector for the four objectives (Pareto frontier). Based on the average vector, we also compute a global average to indicate an overall result of the random approach. For the PostgreSQL model, since it has 86 mutated versions, we perform this process for each mutated version, respectively, to collect result data. Then, we have 86 resulting average vectors and 86 global averages for the 86 mutated versions of the PostgreSQL model.

Fig. 3 illustrates the basic process for each model to collect and compute results from the Pareto solution set obtained by our bandit heuristic search strategy and its optimization with jMetal. For each model, the Pareto solution set has 50 solution candidates, and each candidate has resulting values obtained for four objectives (Pareto frontier) using the 40 fixed seeds. Therefore, for each candidate, we compute the average result vector and the global average by applying the same process as for the random approach. Then, for the Pareto solution set, we have 50 average result vectors and 50 global averages in total. Also, for the PostgreSQL model, we first perform this process for its 86 mutated versions. For each mutated version, we then compute an global average result from the 50 average vectors, and an overall average result from the 50 global averages as the final result for this mutated version.

C. Analysis of Results

We visualize the result data collected using the processes discussed in Fig. 2 and Fig. 3 using box plots. Each box in the plot shows the range between the first and third quartiles

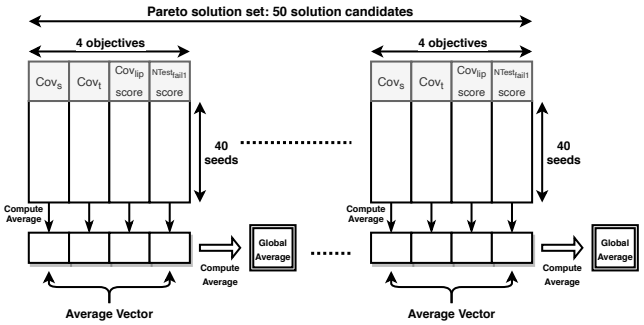


Fig. 3: Result computations for the bandit heuristic search approach

(Q1 and Q3) as a rectangle, with a solid red line indicating the median value. The distance between Q1 and Q3 is the inter-quartile range (IQR); 25 % of the data lies below Q1, and 75 % of the data lies below Q3. The blue dashed lines indicate the smallest (largest) observed point from the dataset that falls within a distance of 1.5 times the IQR below Q1 (and above Q3, respectively). Circles indicate outliers that lie outside 1.5 times the IQR.

Fig. 4 and Fig. 5 shows the box plots for the Java Server Socket and Array List models of the training set. The result data are collected directly when jMetal finish all generations of the NSGA-II algorithm. The size of the resulting dataset to generate each box plot for the random approach is 40 (obtained from 40 seeds shown in Fig. 2), while for the heuristic approach the size of the dataset is 50 (obtained from 50 average vectors shown in Fig. 3).

From Fig. 4 and Fig. 5, we can observe that for the box plots of the Java Server Socket model, our bandit heuristic approach gives better results on the transition coverage Cov_t and the score of $NTest_{fail1}$ compared to the random approach. Concerning state coverage Cov_s and the score of Cov_{lip}, the random approach is slightly better. From the box plots of the Array List model in Fig. 4 and Fig. 5, it can be observed that

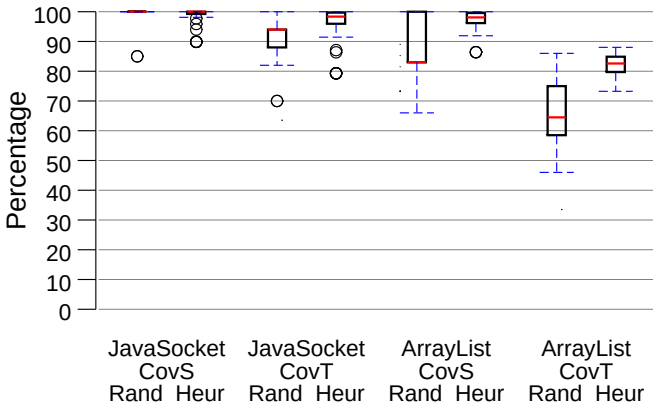


Fig. 4: Comparison of state and transition coverages for Java server socket and array list models

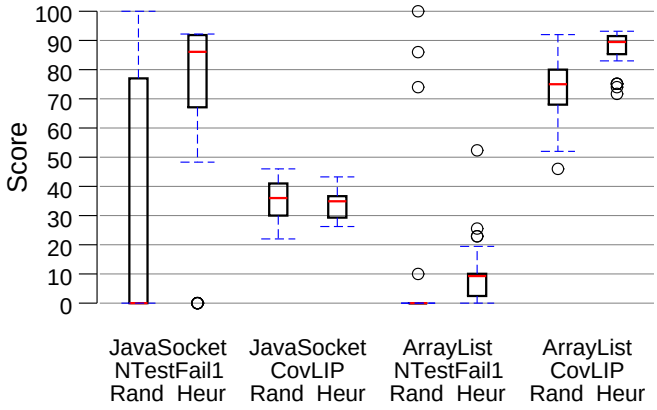


Fig. 5: Comparison of scores to number of test cases used to find the first failure and LIP coverage for Java server socket and array list models

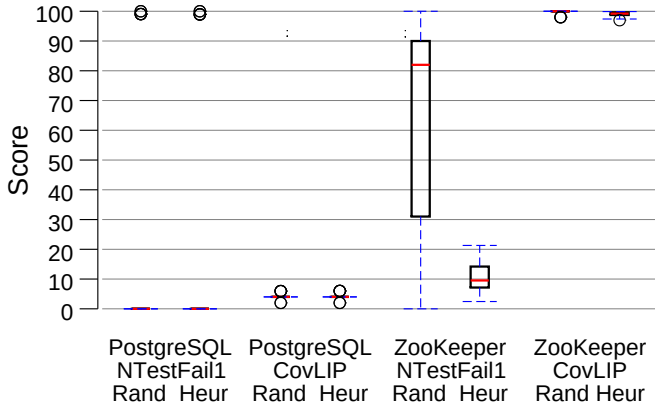


Fig. 7: Comparison of scores to number of test cases used to find the first failure and LIP coverage for PostgreSQL and ZooKeeper models

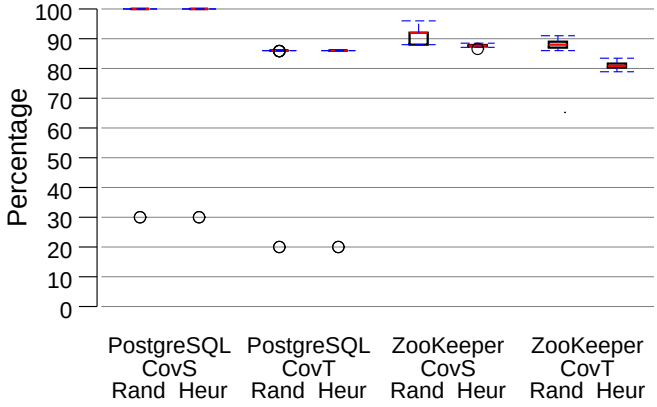


Fig. 6: Comparison of state and transition coverages for PostgreSQL and ZooKeeper models

our bandit heuristic approach has better performance on all objectives in comparison to the random approach. The box plots for both the Linked List and ChooseTest models show that the heuristic approach has better results on all objectives, which is similar to the results of Array List model. Hence, we do not specifically discuss their box plots here due to the space limitations.

After we obtained the Pareto optimal set from the training phase of the four simple models in the training set, we apply the resulting values of eight different rewards in the Pareto optimal set to the 86 mutated PostgreSQL and ZooKeeper models, respectively, from the test set. Fig. 6 and Fig. 7 show the collected data for the PostgreSQL and ZooKeeper models as box plots. The size of the dataset is 86 for the PostgreSQL model with the heuristic approach, since we collected results from 86 mutated versions of the PostgreSQL model. For each version, we collect an average result for each objective from 50 average vectors as was shown in Fig. 3. For the random approach of the PostgreSQL model, the size of the dataset is also 86. This dataset has 86 average vectors, and each vector is computed from the results of 40 seeds as was shown in Fig. 2. For the ZooKeeper model, the sizes of datasets for the heuristic and random approaches are the same as the datasets for the four simple models in the training set.

From the box plots for the PostgreSQL model, we see that the heuristic approach is slightly better than random approach, since the box plot of the transition coverage Cov_t of the heuristic approach does not have the extra outlier (around 85) shown in the plot of the random approach in Fig. 6.

For the resulting box plots of the ZooKeeper model in Fig. 6 and Fig. 7, although the box plots of the random approach seems to have better results on the four objectives than the heuristic approach, the box plots also show that the distribution of the resulting data for heuristic approach is more concentrated than the random approach. For instances, the resulting box plot of the $NTest_{fail1}$ score for the random approach has some extremely bad results (0) and extremely good results (100), compared to the box plot of the heuristic approach. The box plots of the heuristic approach from other models also reflect this characteristic, i. e., they have a more concentrated distribution of their resulting data than random approach.

D. Discussion

Our box plots compare the performance between our bandit heuristic approach and a random approach for each objective separately. To compare all values at a glance, Table II shows the *global average* over all four metrics across all generated tests, as obtained by the process shown in Fig. 2 and Fig. 3. For PostgreSQL, we additionally average the fault-detection rate over 86 mutants [25] on the code.

For the training set, we can see a large improvement on the results both in the best and in the average case, which shows that our heuristic adapts well to four different types of models and produces consistent results. For the test set (PostgreSQL and ZooKeeper), the difference is less clear. The heuristic approach for PostgreSQL has better transition coverage, but the difference is not significant, and the average scores even

Table II: Comparison of global averages obtained by heuristic and random approaches for each model.

Model	Heuristic		Random
	Max GA	Aver GA	GA
ChooseTest	76.81	61.11	50.80
JavaServerSocket	82.26	75.38	64.35
ArrayList	82.62	68.95	59.15
LinkedList	71.19	69.77	58.06
PostgreSQL	72.74	48.45	48.45
ZooKeeper	72.91	69.55	84.79

match up to two digits after the decimal point. For ZooKeeper, the random search does better than the heuristic approach.

Therefore, the results of the bandit heuristic search for the test set are not as good as the results from the training set. The reason for this are that our training set is too small, resulting in overfitting. Even so, the results of the PostgreSQL model from the test set also show a potential for the heuristic approach to perform much better than the random approach.

V. Related Work

A. Test Generation with Multi-objective Optimization

Test generation related to multi-objective optimization using Pareto-effective approaches have been developed in the existing literature. Oster and Saglietti [26] proposed a technique for test data generation using multi-objective optimization and evolutionary algorithms to handle two objectives including data flow coverage and the number of test cases required. They used two variants of genetic algorithms, including the Multi-Objective Aggregation (MOA) and classical NSGA to compare with a random approach and a simulated annealing algorithm, in order to test object-oriented programs implemented in Java. The results showed that simulated annealing outperformed other algorithms, but NSGA offered a higher flexibility since it can provide a complete solution set instead of a single optimal result.

A multi-objective approach to test data generation was presented by Harman et al. [27], which focused on applying multi-objective optimization to branch coverage and generating branch adequate test sets for branch adequate testing. Two objectives were considered by the authors, including branch coverage and dynamic memory consumption. The authors compared the effectiveness of three search approaches: a random, a weighted genetic algorithm search, and the NSGA-II. The case studies were carried out on testing C code from both real-world and synthetic examples.

In this paper, we define four test adequacy criteria as objectives. Instead of branch coverage, we consider path coverage as one of our test adequacy criteria, since path coverage is a stronger test adequacy criterion than branch coverage and it concerns a sequence of branch decisions instead of only one branch at a time. Also, our approach is based on models of the system under test. Path coverage and other criteria are optimized at the model level rather than coverage of the SUT code. We notice that the process of test case generation faces the exploration versus exploitation dilemma, so we propose the heuristic search strategy to handle this dilemma and guide the test case generation. We then use the jMetal framework with the genetic algorithm NSGA-II to tune the strategy in order to optimize the four objectives we defined, with respect to their trade-offs. For the experimental evaluation, we also compare our approach with a random approach.

B. Search-based Test Generation

Our bandit-based heuristic approach also relates to some extent to work on search-based testing, where random testing is augmented with heuristics to find fault-revealing test cases more efficiently [28]. In random testing, the problem of choosing suitable input with the right values and types exists; these problems are taken care of in model-based testing because the user provides a model that generates these inputs. Similarities exist in three of the six heuristics used in Guided Random Testing [28]: 1) Impurity: We use a different weight for self-loop transitions, which contain at least some impure methods as not to be completely redundant; 2) Bloodhound: We also choose transitions based on coverage, but we use coverage at the model level rather than coverage of the SUT code; 3) Orienteering: At this point, we do not consider the time it takes to execute a transition, because the execution times of transition actions did not differ in major ways in our examples.

In addition to using Pareto-efficient approaches for search-based test generation, Salahirad et al. [29] discussed different fitness functions for white-box testing. Their findings confirm that high (source code) coverage is a prerequisite for successful fault detection, and that branch coverage stands out as the most effective single criterion. They also used *treatment learning* to discover which metrics best predicts fault detection. We have not investigated how different subsets of our criteria (especially when used within a limited resource budget) compare to each other, as we only have four, and hence much less than they had to consider. Rojas et al. [30] found that multi-objective optimization algorithms based on Pareto dominance are less suitable than a linear combination of the different non-conflicting objectives. It is not obvious to us how we would prioritize weights among the four different objectives, a question which also [30] left for their future work.

C. Test Section with Multi-objective Optimization

Related work also exists in multi-objective optimization for test selection. Yoo and Harman [31] presented a multi-objective formulation of the regression test case selection problem to show how multiple objectives can be optimized using a Pareto efficient approach. Their goal was to find a subset of a test suite, which is a Pareto optimal set with respect to the chosen test criteria. They gave three algorithms to compare their effectiveness, including a single-objective greedy algorithm, NSGA-II, and vNSGA-II. The evaluation was carried out on five programs in the Siemens suite and a program from the European Space Agency. For each program, the authors randomly selected test suites from existing available test suites as the input to the multi-objective optimization

process. The results showed that the NSGA-based approaches can out-perform the greedy approach.

Mondal et al. [32] studied multi-objective test case selection to analyze both coverage-based and diversity-based test case selection approaches. They proposed two approaches for bi- and three-objectives optimization, respectively, by considering code coverage, test case diversity, and test execution time. They used the Additional-Greedy and NSGA-II algorithms for bi-objective optimization and NSGA-II only for three-objective optimization on 16 versions of five real-world programs, such as JBoss and Apache Ant. The results showed an improvement of the fault detection rate by the three-objective optimization approach.

For our work, we do not have existing available test suites, so we focus on using the bandit-based heuristic search to generate optimal test cases directly, with the aid of the Pareto-efficient approach to optimize the four test adequacy criteria. Also, instead of considering test execution time as an objective, we use the number of test cases to find the first failure together with three different coverages as objectives.

VI. Conclusions and Future Work

Our main contribution is a heuristic search based test case generation approach for model-based testing, aiming at performing well on test adequacy criteria with considering their trade-offs. We have proposed a bandit-based heuristic search strategy to handle the exploration versus exploitation dilemma for test case generation. Then, we applied an optimization technique to tune our strategy and optimize the chosen test adequacy criteria with the aid of the jMetal multi-objective optimization framework and the NSGA-II Pareto-efficient multi-objective genetic algorithm. We have evaluated our approach on several models for the Modbat model-based testing tool by comparing the results of the bandit-based heuristic search with a random test case generation approach. Our experiments show that our bandit-based heuristic search approach has potential to obtain better and more consistent results on the chosen adequacy criteria compared to random test case generation, while considering the trade-offs of the test adequacy criteria.

The second contribution is an implementation of the bandit-based heuristic search strategy in the Modbat. This implementation is now included as a new feature in the Modbat 3.4 release. We have defined test adequacy criteria as multi-objectives so that Modbat implements our strategy for test case generation in addition to its standard random search. The test adequacy criteria are optimized using the jMetal framework which applies the NSGA-II algorithm on the Modbat models that we use as the training set to find a Pareto optimal set. The reward parameter settings obtained in the Pareto optimal set can then be used to initialize the test case generation with our bandit-based heuristic search strategy to generate test cases for advanced Modbat models in general and targeting the chosen test adequacy criteria.

The work presented in this paper opens up several directions of future work. The results of our heuristic search from the training set are promising, but the results from the test set are not as good, especially for the ZooKeeper model, due to overfitting. To leverage the potential of our approach, we need to obtain more and more diverse models for the training set so that multi-objective optimization can get well-fitted reward parameter settings in Pareto optimal set. Additionally, increasing the size of the population and the number of generations for the NSGA-II might give us better results in the Pareto optimal set. Also, we may consider using alternative algorithms provided by jMetal to solve our multi-objective optimization problem.

Another direction of future work is to investigate self-optimizing approaches with optimization on the models at runtime to achieve the potential of our bandit heuristic search strategy for test case generation. Furthermore, other test adequacy criteria such as the execution time of test cases could be considered as additional objectives for the optimization. Also, in addition to bandit heuristic search strategy, we plan to implement other heuristic strategies for test case generation. Finally, the application of our approach to platforms other than Modbat is another possibility.

Acknowledgements

This work was partially supported by the European Horizon 2020 project COEMS under grant agreement no. 732016 (https://www.coems.eu/).

References

[1] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Software testing, verification and reliability*, vol. 22, no. 5, pp. 297–312, 2012.

[2] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[3] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, "A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II," in *International conference on parallel problem solving from nature*. Springer, 2000, pp. 849–858.

[4] C. Artho, A. Biere, M. Hagiya, E. Platon, M. Seidl, Y. Tanabe, and M. Yamamoto, "Modbat: A model-based API tester for event-driven systems," in *Haifa Verification Conference*, ser. Lecture Notes in Computer Science, vol. 8244. Springer, 2013, pp. 112–128.

[5] "Modbat 3.4," https://github.com/cyrille-artho/modbat/tree/3.4.

[6] J. J. Durillo and A. J. Nebro, "jMetal: A Java framework for multi-objective optimization," *Advances in Engineering Software*, vol. 42, no. 10, pp. 760 – 771, 2011.

[7] A. J. Nebro, J. J. Durillo, and M. Vergne, "Redesigning the JMetal multi-objective optimization framework," in *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO Companion '15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 1093–1100.

[8] K.-T. Cheng and A. S. Krishnakumar, "Automatic functional test generation using the extended finite state machine model," in *30th ACM/IEEE Design Automation Conference*. IEEE, 1993, pp. 86–91.

[9] Programming Methods Laboratory of École Polytechnique Fédérale de Lausanne, "The Scala Programming Language," https://www.scala-lang.org.

[10] R. Wang, C. Artho, L. M. Kristensen, and V. Stolz, "Visualization and abstractions for execution paths in model-based software testing," in *Integrated Formal Methods*, ser. Lecture Notes in Computer Science, W. Ahrendt and S. L. Tapia Tarifa, Eds., vol. 11918. Springer, 2019, pp. 474–492.

[11] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. Cambridge, MA: MIT Press, 2011.

[12] D. A. Berry and B. Fristedt, "Bandit problems: sequential allocation of experiments (monographs on statistics and applied probability)," *London: Chapman and Hall*, vol. 5, pp. 71–87, 1985.

[13] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of Monte Carlo tree search methods," *IEEE Trans. on Comput. Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.
[14] V. Kuleshov and D. Precup, "Algorithms for multi-armed bandit problems," *arXiv preprint arXiv:1402.6028*, 2014.
[15] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, no. 2-3, pp. 235–256, 2002.
[16] M. Tokic and G. Palm, "Value-difference based exploration: adaptive control between epsilon-greedy and softmax," in *Annual Conference on Artificial Intelligence*. Springer, 2011, pp. 335–346.
[17] S. Gelly and D. Silver, "Monte-Carlo tree search and rapid action value estimation in computer Go," *Artificial Intelligence*, vol. 175, no. 11, pp. 1856–1875, 2011.
[18] T. L. Lai and H. Robbins, "Asymptotically efficient adaptive allocation rules," *Advances in applied mathematics*, vol. 6, no. 1, pp. 4–22, 1985.
[19] Y. Collette and P. Siarry, *Multiobjective optimization: principles and case studies*. Springer Science & Business Media, 2013.
[20] M. Ehrgott, *Multicriteria optimization*. Springer Science & Business Media, 2005.
[21] C. A. C. Coello, G. B. Lamont, D. A. Van Veldhuizen *et al.*, *Evolutionary algorithms for solving multi-objective problems*. Springer, 2007.
[22] C. Artho and G. Rousset, "Model-based testing of the Java network API," *arXiv preprint arXiv:1703.07034*, 2017.
[23] C. Artho, M. Seidl, Q. Gros, E. Choi, T. Kitamura, A. Mori, R. Ramler, and Y. Yamagata, "Model-based testing of stateful APIs with Modbat," in *Proc. 30th Intl. Conf. on Automated Software Engineering (ASE 2015)*. Lincoln, USA: IEEE, Nov 2015, pp. 858–863.
[24] C. Artho, Q. Gros, G. Rousset, K. Banzai, L. Ma, T. Kitamura, M. Hagiya, Y. Tanabe, and M. Yamamoto, "Model-based API testing of Apache ZooKeeper," in *2017 IEEE Intl. Conf. on Software Testing, Verification and Validation (ICST)*. IEEE, 2017, pp. 288–298.
[25] D. Tziatzios, "Model-based testing for SQL databases," Master's thesis, KTH, School of Electrical Engineering and Computer Science (EECS), 2019.
[26] N. Oster and F. Saglietti, "Automatic test data generation by multi-objective optimisation," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2006, pp. 426–438.
[27] K. Lakhotia, M. Harman, and P. McMinn, "A multi-objective approach to search-based test data generation," in *Proc. of the 9th annual conference on Genetic and evolutionary computation*. ACM, 2007, pp. 1098–1105.
[28] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler, "GRT: program-analysis-guided random testing," in *Proc. 30th Intl. Conf. on Automated Software Engineering (ASE 2015)*. Lincoln, USA: IEEE, Nov 2015, pp. 212–223.
[29] A. Salahirad, H. Almulla, and G. Gay, "Choosing the fitness function for the job: Automated generation of test suites that detect real faults," *Softw. Test., Verif. Reliab.*, vol. 29, no. 4-5, 2019.
[30] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri, "Combining multiple coverage criteria in search-based unit test generation," in *Search-Based Software Engineering*, M. Barros and Y. Labiche, Eds. Springer, 2015, pp. 93–108.
[31] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *Proceedings of the 2007 international symposium on Software testing and analysis*, 2007, pp. 140–150.
[32] D. Mondal, H. Hemmati, and S. Durocher, "Exploring test suite diversification and code coverage in multi-objective test case selection," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2015, pp. 1–10.