Check for updates

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

© Springer Nature Switzerland AG 2018
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.

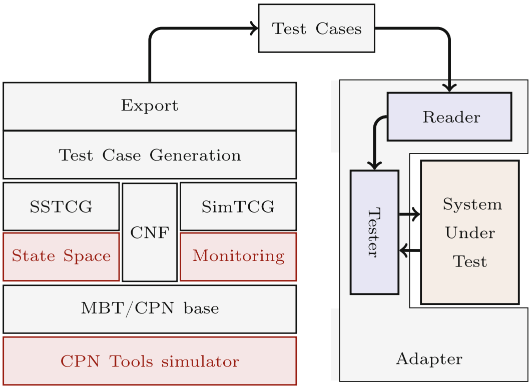


Fig. 1. Overview of MBT/CPN modules.

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.

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.

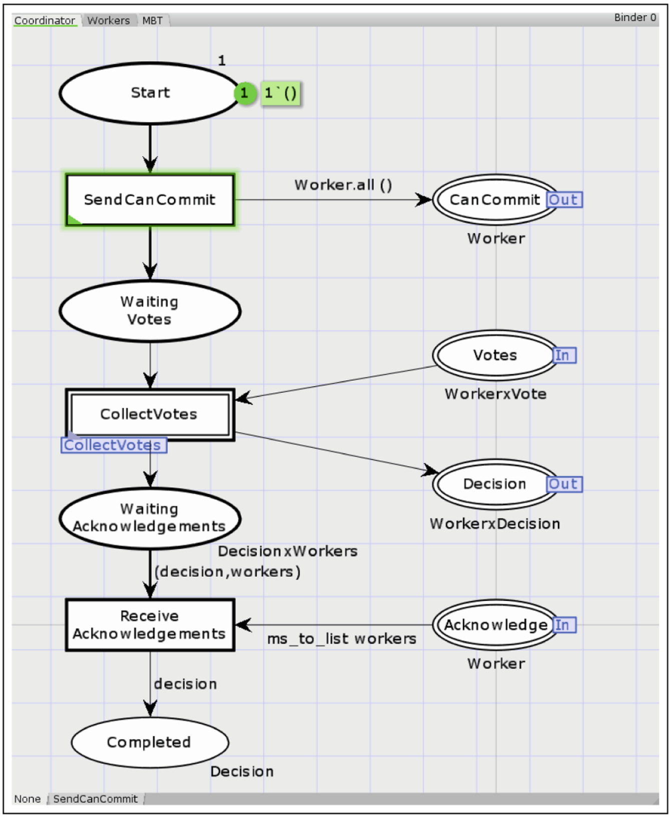


Fig. 4. MBT/CPN example in CPN Tools: Coordinator module.

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

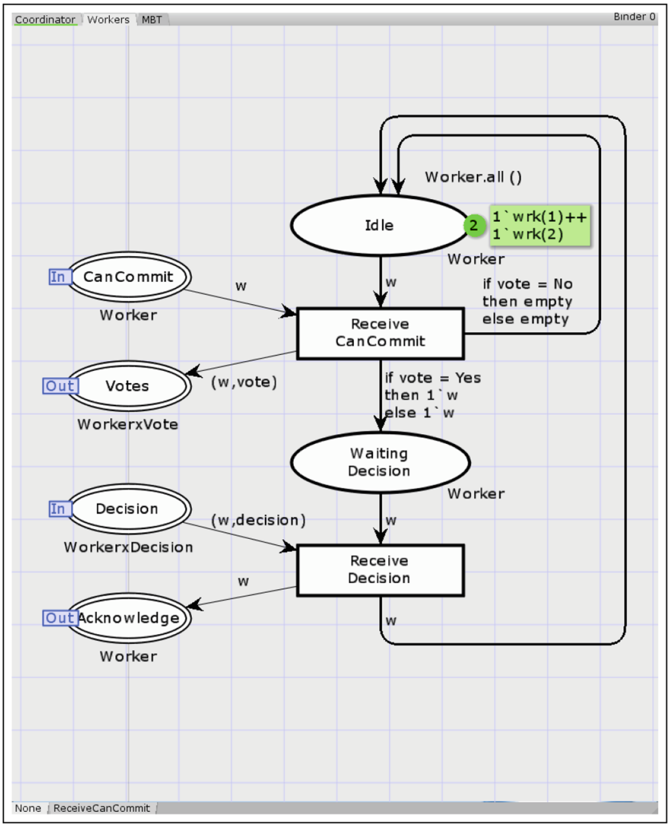
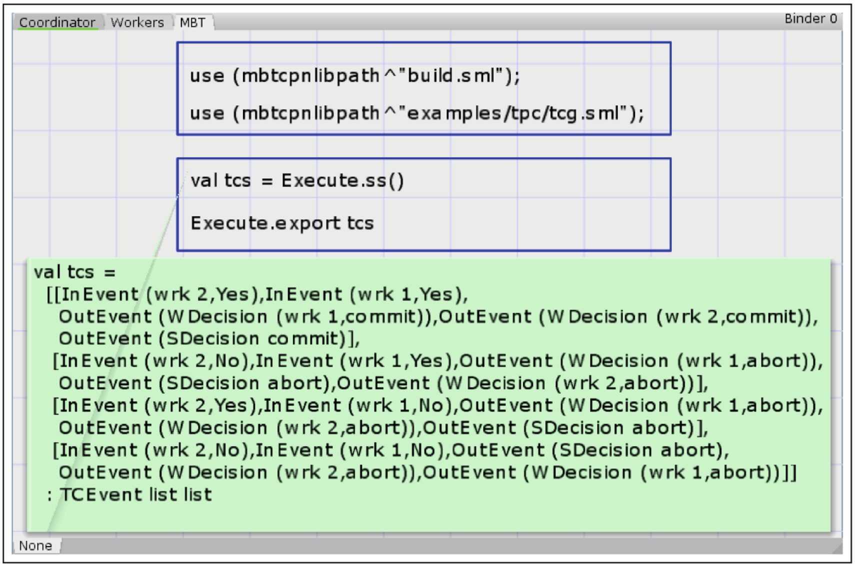


Fig. 5. MBT/CPN example in CPN Tools: Workers module.



```
use (mbtcpnlibpath^"build.sml");

use (mbtcpnlibpath^"examples/tpc/tcg.sml");
```

```
val tcs = Execute.ss()

Execute.export tcs
```

```
val tcs =
  [[InEvent (wrk 2,Yes),InEvent (wrk 1,Yes),
    OutEvent (WDecision (wrk 1,commit)),OutEvent (WDecision (wrk 2,commit)),
    OutEvent (SDecision commit)],
   [InEvent (wrk 2,No),InEvent (wrk 1,Yes),OutEvent (WDecision (wrk 1,abort)),
    OutEvent (SDecision abort),OutEvent (WDecision (wrk 2,abort))],
   [InEvent (wrk 2,Yes),InEvent (wrk 1,No),OutEvent (WDecision (wrk 1,abort)),
    OutEvent (WDecision (wrk 2,abort)),OutEvent (SDecision abort)],
   [InEvent (wrk 2,No),InEvent (wrk 1,No),OutEvent (SDecision abort),
    OutEvent (WDecision (wrk 2,abort)),OutEvent (WDecision (wrk 1,abort))]]
  : TCEvent list list
```

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$^\sharp$ 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)