Innovations in Systems and Software Engineering
https://doi.org/10.1007/s11334-023-00528-z

S.I. : VECOS 2021

Coverage visualization and analysis of net inscriptions in coloured Petri net models

Faustin Ahishakiye[1] · José Ignacio Requeno Jarabo[1,2] · Lars Michael Kristensen[1] · Volker Stolz[1]

Received: 12 April 2022 / Accepted: 15 March 2023
© The Author(s) 2023

Abstract

High-level Petri nets such as coloured Petri nets (CPNs) are characterized by the combination of Petri nets and a high-level programming language. In CPNs and CPN Tools, the inscriptions (e.g. arc expressions and guards) are specified using Standard ML. The application of simulation and state space exploration for validating CPN models traditionally focuses on behavioural properties related to net structure, i.e. places and transitions. This means that the net inscriptions are only implicitly validated, and the extent to which their sub-expressions have been covered is not made explicit. This paper extends our previous work on coverage analysis of net inscriptions of CPN models. In particular, we improve the CPN Tools library responsible for annotating, instrumenting and collecting the evaluation of Boolean conditions for determining the coverage criteria based on model executions. The library now automates most of the instrumentation parts that were done manually before and integrates the reports of the coverage analysis into the CPN Tools GUI. We evaluate our approach on new publicly available CPN models.

Keywords Coverage analysis · MC/DC · CPN model · Testing

1 Introduction

Coverage analysis is important for programs in relation to fault detection. Structural coverage criteria are required for software safety and quality design assurance [1], and low coverage indicates that the software product has not been extensively tested. Two common metrics are statement and branch coverage [2], where low coverage concretely indicates that certain instructions have never actually been executed. Coloured Petri nets [3] and CPN Tools [4] have been widely used for constructing models of concurrent systems with simulation and state space exploration (SSE) being the two main techniques for dynamic analysis. CPN model analysis is generally concerned with behavioural properties related to boundedness, reachability, liveness and fairness properties. This means that the main focus is on structural elements such as places, tokens, markings (states), transitions and transition bindings. Arc expressions and guards are only implicitly considered via the evaluation of these net inscriptions taking place as part of the computation of transition enabling and occurrence during model execution. This means that design errors in net inscriptions may not be detected as we do not obtain explicit information on for instance whether both branches of an if–then–else expression on an arc have been covered.

We argue that from a software engineering perspective, it is important to be explicitly concerned with quantitative and qualitative analysis of the extent to which net inscriptions have been covered. Our hypothesis is that the coverage criteria used for traditional source code can also be applied to the net inscriptions of CPN models. Specifically, we consider

José Ignacio Requeno Jarabo, Lars Michael Kristensen and Volker Stolz have contributed equally to this work.

✉ Faustin Ahishakiye
faahi7267@uib.no

José Ignacio Requeno Jarabo
jrequeno@ucm.es

Lars Michael Kristensen
lmkr@hvl.no

Volker Stolz
vsto@hvl.no

1 Department of Computer Science, Electrical Engineering, and Mathematical Sciences, Western Norway University of Applied Sciences, Inndalsveien 28, 5063 Bergen, Norway

2 Information Systems and Computing, Complutense University of Madrid, C/Prof. José García Santesmases, 9, 28040 Madrid, Spain

Published online: 25 May 2023

the modified condition decision coverage (MC/DC) criterion. MC/DC is a well-established coverage criterion for safety-critical systems and is required by certification standards, such as the DO-178C [5] in the domain of avionic software systems. In the context of MC/DC, a *decision* is a Boolean expression composed of sub-expressions and Boolean connectives (such as logical conjunction). A *condition* is an atomic (Boolean) expression. According to the definition of MC/DC [2, 6], each condition in a decision has to show an independent effect on that decision's outcome by: (1) varying just that condition while holding fixed all other possible conditions or (2) varying just that condition while holding fixed all other possible conditions that could affect the outcome. MC/DC is a coverage criterion at the condition level and is recommended due to its advantages of being sensitive to code structure, requiring few test cases ($n + 1$ for n conditions), and it is the only criterion that considers the independence effect of each condition.

Coverage analysis for software is usually provided through dedicated instrumentation of the software under test, by either the compiler or additional tooling, such as binary instrumentation. Transferring this to a CPN model under test, our aim is to combine the execution of a CPN model (by simulation or SSE) with coverage analysis of SML guard and arc expressions. Within CPN Tools, there is no coverage analysis of the SML expressions in a CPN model. This means that to record coverage data for a CPN model under test, it is necessary to instrument the Boolean expressions such that the truth values of individual conditions are logged in addition to the overall outcome of the decision. Our approach to instrumentation makes use of side effects by outputting intermediate results of conditions and decisions, which we then process to obtain the coverage verdict. No modifications to the net structure of the CPN model are necessary. Furthermore, the instrumentation has little impact on model execution so that it does not delay the simulation and SSE.

In this article, we extend our approach for coverage analysis of net inscriptions in CPN models [7] with the following new contributions:

1. We automate our instrumentation: it takes as input the original CPN model and produces an instrumented model where the Boolean expressions in guards and arcs are transformed into a form that emits log entries that are collected for coverage analysis. The automatic instrumentation also processes the definition of auxiliary SML functions, which were not considered in our manually instrumented solution.
2. We integrated a coverage visualization in CPN model such that a tester can observe which guard and arc expressions are covered or not. The covered parts are highlighted in green, whereas the uncovered parts are shown in red with a possibility to see coverage percentage for each decision.
3. We test more CPN models and gather coverage statistics for seven additional CPN models publicly available.

The remainder of this paper is organized as follows. In Sect. 2, we introduce the MC/DC coverage criterion in more detail. In Sect. 3, we present our approach to deriving coverage data and show how to instrument guard and arc expressions to collect the required coverage data. In Sect. 4 we consider the post-processing of coverage data. We demonstrate the application of our library for coverage analysis on publicly available CPN models in Sect. 5. In this section, we also evaluate our approach with respect to overhead in execution and discuss our findings. Section 6 discusses related work, and we present our conclusions including directions for future work in Sect. 7. Our coverage analysis library with the instrumented example models, the Python code to instrument and produce reports and graphs, and documentation is available at https://github.com/selabhvl/cpnmcdctesting [8].

2 Coverage analysis and MC/DC

When considering CPN models, we will be concerned with coverage analysis of guard and arc coverage of expressions. A guard expression is a list of Boolean expressions all of which are required to evaluate to true in a given transition binding for the transition to be enabled. We refer to such Boolean expressions as *decisions*. Similarly, an if–then–else expression on an arc will have a decision determining whether the then or the else branch will be taken. Decisions are constructed from *conditions* and Boolean operators.

Definition 1 (Condition, Decision) A **condition** is a Boolean expression containing no Boolean operators except for the unary operator NOT.

A **decision** is a Boolean expression composed of conditions and zero or more Boolean operators. It is denoted by $\mathrm{D}(c_1, c_2, \cdots, c_i, \cdots, c_n)$, where c_i, $1 \leq i \leq n$ are conditions.

As an example, we may have a guard (or an arc expression) containing a decision of the form $D = (a \wedge b) \vee c$, where a, b and c are conditions. These conditions may in turn refer to the values bound to the variables of the transition. The evaluation of a decision requires a *test case* assigning a $value \in \{0, 1, ?\}$ to the conditions of the decision, where ? means that a condition was not evaluated due to short-circuiting. Short circuit means that the right operand of the *and*-operator (&&/$\wedge$) is not evaluated whether the left operand is false, and the right operand of the *or*-operator (||/$\vee$) is not evaluated whether the left operand is true.

Depending on the software safety level (A-D) which is assessed by examining the effects of a failure in the system, different structure coverage criteria are required: *statement* coverage for software levels A-C, *branch/decision* coverage for software levels A-B and MC/DC for software level A [2]. Statement coverage is considered inadequate because it is insensitive to some control structures. Both statement and branch coverage are completely insensitive to the logical operators ($\vee$ and $\wedge$) [9]. The criteria taking logical expressions into consideration have been defined [1]. These are *condition coverage* (CC), where each condition in a decision takes on each possible outcome at least once true and once false during testing; *decision coverage* (DC) requiring only each decision to be evaluated once true and once false; and *multiple condition coverage* (MCC) which is an exhaustive testing of all possible input combinations of conditions to a decision. CC and DC are considered inadequate due to ignorance of the independence effect of conditions on the decision outcome. MCC requires 2^n tests for a decision with n inputs. This results in exponential growth in the number of test cases and is therefore time-consuming and impractical for many test cases.

To address the limitations of the coverage criteria discussed above, *modified condition/decision coverage* (MC/DC) is considered and is required for safety-critical systems such as in the avionics industry. MC/DC has been chosen as the coverage criterion for the highest safety-level software because it is sensitive to the complexity of the decision structure [6] and requires only $n + 1$ test cases for a decision with n conditions [1, 10]. In addition, MC/DC coverage criterion is suggested as a good candidate for model-based development (MBD) using tools such as Simulink and SCADE [11]. Thus, our model coverage analysis is based on MC/DC criterion. The following MC/DC definition is based on DO-178C [2]:

Definition 2 (Modified condition/decision coverage) A program is MC/DC covered and satisfies the MC/DC criterion if the following holds:

- Every point of entry and exit in the program has been invoked at least once,
- Every condition in a decision in the program has taken all possible outcomes at least once,
- Every decision in the program has taken all possible outcomes at least once,
- Each condition in a decision has shown to independently affect that decision's outcome by: (1) varying just that condition while holding fixed all other possible conditions or (2) varying just that condition while holding fixed all other possible conditions that could affect the outcome.

Table 1 MCC and selected MC/DC test cases for decision $D = (a \wedge b) \vee c$

TC	a	b	c	D	MC/DC pairs
(a) MCC test cases					
1	0	0	0	0	
2	0	0	1	1	c(1,2)
3	0	1	0	0	
4	0	1	1	1	c(3,4)
5	1	0	0	0	
6	1	0	1	1	c(5,6)
7	1	1	0	1	a(3,7), b(5,7)
8	1	1	1	1	
(b) Selected MC/DC test cases					
1	0	?	0	0	
2	1	1	?	1	a(1,2)
3	1	0	0	0	b(2,3)
4	0	?	1	1	c(1,4)

The coverage of program entry and exit in Definition 2 is added to all control flow criteria and is not directly connected with the main point of MC/DC [12]. The most challenging and discussed part is showing the independent effect, which demonstrates that each condition of the decision has a defined purpose. The item (1) in the definition defines the unique cause MC/DC and item (2) has been introduced in the DO-178C to clarify that the so-called *Masked MC/DC* is allowed [5, 13]. Masked MC/DC means that it is sufficient to show the independent effect of a condition by holding fixed only those conditions that could actually influence the outcome. Thus, in our analysis, we are interested in evaluation of expressions by checking the independence effect of each condition.

Example 1 Consider the decision $D = (a \wedge b) \vee c$. Table 1a presents all eight possible test cases (combinations) for MCC. The MC/DC pairs column for example, $c(1, 2)$ specifies that from test case 1 and 2 we can observe that changing the truth value of c while keeping the values of a and b, we can affect the outcome of the decision. Comparing MCC to MC/DC in terms of the number of test cases, there are seven possible MC/DC test cases (1 through 7) that are part of an MC/DC pair, where condition c is represented by three MC/DC pairs of test cases. However, for a decision with three conditions, only four (i.e. $n + 1$) test cases are required to achieve MC/DC coverage as shown in Table 1b, where '?' represents the condition that was not evaluated due to short-circuiting.

3 Instrumentation of CPN models

In this section, we describe our instrumentation approach on an example CPN model and highlight the salient features of

our coverage analysis library. Our overall goal is that through simulation or SSE, we instrument and (partially) fill a truth table for each decision in the net inscriptions of the CPN model. Then, for each of these tables, and hence the decisions they are attached to, we determine whether the model executions that we have seen so far satisfy the MC/DC coverage criteria. If MC/DC is not satisfied, either further simulations are necessary, or if the state space is exhausted, developers need to consider the reason for this shortcoming, which may be related to insufficient exploration as per a limited set of initial markings, or a conceptual problem in that certain conditions indeed cannot contribute to the overall outcome of the decision.

3.1 MC/DC coverage for CPN models

MC/DC coverage (or any other type of coverage) is commonly used with executable programs: which decisions and conditions were evaluated by the test cases, and with which result. Specifically, these are decisions *from the source code* of the system (application) under test. Of course, a compiler may introduce additional conditionals into the code during code generation, but these are not of concern. CPN Tools already reports a primitive type of coverage as part of simulation (the transition and transition bindings that have been executed) and the state space exploration (transitions that have never occurred). These can be interpreted as variants of state and branch coverage.

Hence, we first need to address what we want MC/DC coverage to mean in the context of CPN models. If we first consider guard expressions on transitions, then we have two interesting questions related to coverage: if there is a guard, we know from the state space (or simulation) report whether the transition has occurred and hence whether the guard expression has evaluated to true. However, we do not know whether during the calculation of enabling by CPN Tools it ever has been false. If the guard had never evaluated to false, this may indicate a problem in the model or the requirements it came from, since apparently that guard was not actually necessary. Furthermore, if a decision in a guard is a complex expression, then as per MC/DC, we would like to see evidence that each condition contributed to the outcome. Neither case can be deduced from the state space report or via the CTL model checker of CPN Tools as the executions only contain transition bindings that have occurred and hence cases where the guard has evaluated to true.

3.2 Automated instrumentation of net inscriptions

In the following, we describe how we instrument the guards on transitions such that coverage data can be obtained. We developed an automated instrumentation based on the `.cpn` XML file of CPN Tools in combination with an SML parser. Arc expressions are handled analogously. Guards in a CPN model are written following the general form of a comma-separated list of Boolean expressions (decisions):

$$[b\mathrm{Exp}_0, \ldots, b\mathrm{Exp}_n]$$

A special case is the expression

$$\mathrm{var} = \mathrm{exp}$$

which may have two effects: if the variable `var` is bound already via a pattern in another expression (arc or guard) of the transition, then this is indeed a Boolean equality test (decision). If, however, `var` is not bound via other expressions, then this assigns the value of `exp` to the variable `var` and does not contribute to any guarding effect.

We consider general Boolean expressions which may make use of the full feature set of the SML language for expressions, most importantly Boolean binary operations, negation, conditional expressions with if–then–else and function calls. Simplified, we handle:

$$\langle bExp \rangle ::= \texttt{not}\ \langle bExp \rangle \mid \langle var \rangle \mid \mathrm{f}\ \langle exp \rangle_0 \ldots \langle exp \rangle_n$$
$$\mid \langle bExp \rangle\ \texttt{andalso}\ \langle bExp \rangle \mid \langle bExp \rangle\ \texttt{orelse}\ \langle bExp \rangle$$
$$\mid \texttt{if}\ \langle bExp \rangle\ \texttt{then}\ \langle bExp \rangle\ \texttt{else}\ \langle bExp \rangle$$
$$\mid \texttt{let} \ldots \texttt{in}\ \langle bExp \rangle\ \texttt{end}$$

Function symbols `f` cover user-defined functions as well as (built-in) relational operators such as $<, =$; we do not detail the overall nature of arbitrary SML expressions, but refer the reader to [14] for a comprehensive discussion. The automatic instrumentation also processes the definition of SML functions in the body of the `.cpn` XML file, which were not considered in our manually instrumented solution. We do not provide instrumentation to measure coverage of pattern matching in function definitions and **`case`** expressions.

SSE or simulation of the model is not in itself sufficient to determine the outcome of the overall expression and its sub-expressions: guards are not explicitly represented, and we only have the event of taking the transition in the state space, but no value of the guard expressions. Hence, we need to rely on side effects during model execution to record the intermediate results. Our key idea is to transform every sub-expression and the overall decision into a form which will use SML's file input/output to emit a log entry that we can collect and analyse. The coverage statistics is calculated from the logged entries through a Python script that is easy to reuse in other contexts.

Listing 1 Expressions

```
datatype condition =
   AND of condition * condition
 | OR of condition * condition
 | NOT of condition
 | ITE of condition * condition
          * condition
 | AP of string * bool;
```

Listing 2 Evaluation function

```
fun eval (AP (cond,v))=([(cond, SOME v)],v)
  | eval (OR (a,b)) = let
val (ares,a') = eval a;
val (bres,b') = eval b;
in
(ares^^bres, a' orelse b')
end
...
fun EXPR (name,expr) : bool = [ ... ]
```

For the necessary instrumentation, a transformation of guard and arc expressions, we essentially create an interpreter for Boolean expressions: when guards are checked (in a deterministic order due to SML's semantics from left to right), we traverse a term representation of the Boolean expression and output the intermediate results. The Boolean expressions that are found in the definition of the SML functions will also trigger log messages during the SSE or simulation of the model.

We have designed a data type (see Listing 1) that can capture the above constructs, and define an evaluation function (see Listing 2) on it. As we later need to map coverage reports back to code, for overall expressions `EXPR` and atomic proposition `AP`, we introduce a component of type string that allows this identification. The evaluation function `eval` collects the result of intermediate evaluations in a list data structure, and the `EXPR` function (implementation not shown) turns this result into a single Boolean value that is used in the guard, and as a side effect outputs the truth value outcome for individual conditions. As an example, if we consider a guard: `a>0 andalso (b orelse (c=42))`; then we can transform this guard in a straightforward manner into `EXPR(''Gid'', AND(AP(''1'', a>0), OR(AP(''2'', b), AP(''3'', c=42))))`. It is important to notice that this does not give us the (symbolic) Boolean expressions, as we still leave it to the standard SML semantics to evaluate the `a>0`, while abstractly we refer to the AP as a condition named "1." We elide expression and proposition names for clarity in the text when not needed.

Any sub-expression must be *total* and not crash and abort the model execution. A short-circuiting evaluation needs to explicitly incorporate the **`andalso`** or **`orelse`** operator and becomes more verbose; hence, for example, `x=0 orelse (y/x >0.0)` becomes
`OR (AP(''O1'', x=0)) (AP(''O2'', y/x > 0.0))`.

We can likewise apply the transformation to Boolean expressions in arc expressions: any Boolean expression is transformed into `EXPR(...(AP ("A`$_n$`",bExp))...)`, resulting for example in the transformation of
`if bexp1 orelse bexp2 then e1 else e2` into
`if EXPR(''E1'',OR(AP (''1'',bexp1), AP (''2'',bexp2))) then e1 else e2`.

Figure 1 shows the sub-modules contained in the Paxos CPN model [15] called the initial Proposer and it is associated to the *InitProposer* substitution transition. It initializes Proposers to obtain a new 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 [15].

The "InitProposer" module is one of several modules of the Paxos model, and the arc and guard expressions in the other modules were transformed in a similar manner. The figure also illustrates how, after evaluating coverage data, we indicate full coverage by colouring the guard green, otherwise red.

4 Post-processing of coverage data

We now discuss the coverage analysis which is performed via post-processing of the coverage data recorded through the instrumentation. We did not implement the MC/DC coverage analysis in SML directly. Rather, we feed individual observations about decision outcomes and their constituent conditions into a Python tool that computes the coverage results. This allows us to reuse the back end in other situations, without being SML or CPN specific.

4.1 Coverage analysis

The general format from the instrumentation step is a sequence of colon-delimited rows, where each triple in a row captures a single decision with the truth values of all conditions in a fixed order and the outcome. As an example, see Script 4.1. The name (stemming from the first argument to an `EXPR` above) is configurable and should be unique in the model; and derived from the name of the element (guard or arc) the expression is attached to. This makes it easy to later trace coverage results for this name back to the element in the model. We recommend to derive the name from the element (guard or arc) the expression is attached to. This makes it easy to later trace coverage results for this name back to the element in the model, and for the user to navigate to the sub-module containing the element should they desire to do so.

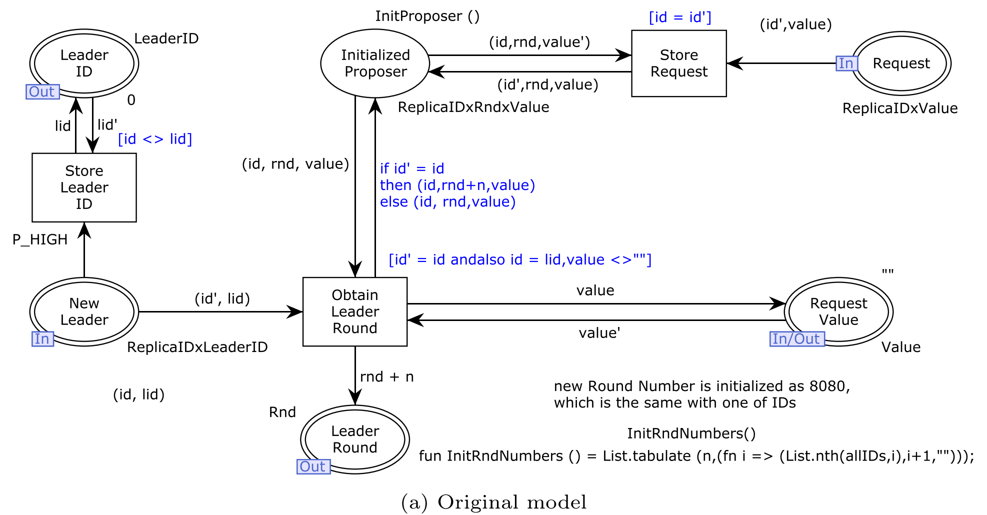
Script 4.1: Log decisions

```
...
a3:01:0
t42:01110:0
t42:01011:1
...
```

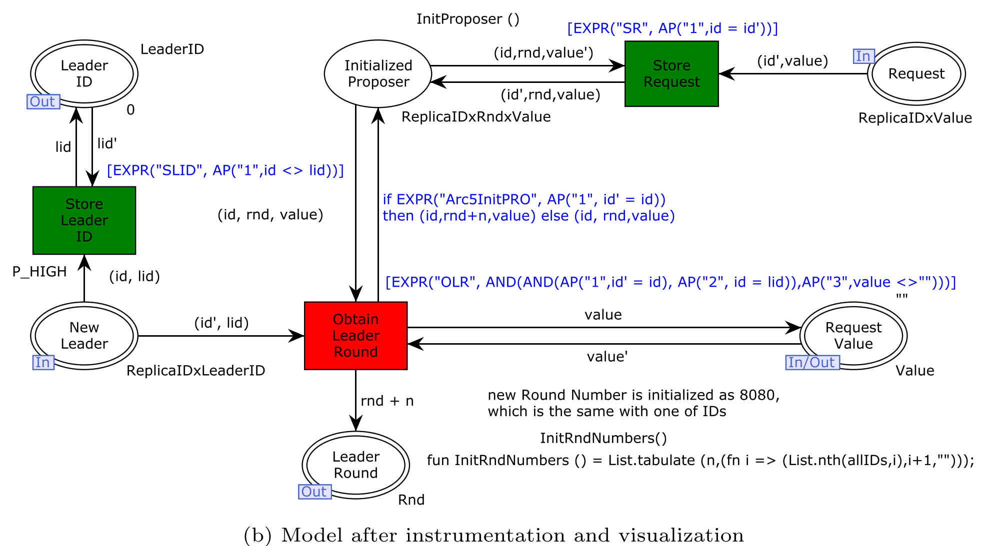
Script 4.2: Decisions evaluation table

```
…
Returna19
0001        0
0010        0
0101        0
0110        0
1001        1
1101        1
1110        1
…
MCDC covered? False
R{1:[(0001, 1001), (0101,
1101), (0110, 1110)], 2:[],
3:[], 4:[]}
```

Fig. 1 Paxos [15]: Guard and arc expressions before and after instrumentation



(a) Original model



(b) Model after instrumentation and visualization

Script 4.1 shows that the decision "$t42$" was triggered twice, possibly on a guard which did not enable the transition (outcome indicating false), after which the exploration choose different transition bindings which resulted in a changed outcome of the 3rd and 5th condition in this decision and an overall outcome of true. We chose to print the binary representation instead of, for example, a slightly shorter integer value to facilitate casual reading of the trace. Also, this allows us to enforce the correct number of bits that we expect per observation, corresponding to the number of conditions in the decision, which mitigates against instrumentation or naming mistakes.

Our Python tool parses the log file and calculates coverage information. It prints the percentage of decisions that are MC/DC and branch covered in textual mode and in GNU Plot syntax (see charts in Fig. 3). The output contains individual reports in the form of the truth tables for each decision, which summarizes the conditions that are fired during the execution of the CPN model, and sets of pairs of test cases per condition that show the independence effect of that condition.

In the case that the decision is not MC/DC covered, the information provided by the Python script helps to infer the remaining valuations of the truth tables that should be evaluated in order to fulfil this criteria. In the example in Script 4.2, the first condition (leftmost column in the table) has multiple complementary entries where the expression only varies in one bit (e.g. rows 0001 and 1001) and the output changes (0 to 1). The `R` set shows three such pairs for condition 1, but no complementary entries at all are found in the truth table for conditions 2, 3 and 4, and hence indicated as empty sets `[]` by Python. This information can then be used by developers to drill down into parts of their model, e.g. through simulation, that have not been covered adequately yet.

4.2 Combining coverage data from multiple runs

Coverage or testing frameworks rely on their correct use by the operator, only a sub-class of tools such as fuzzers are completely automated. Our central `mcdcgen()` function only explores the state space for the current configuration as determined by the initial markings. Compared to regular testing of software, this corresponds to providing a single input to the system under test.

It is straightforward to capture executions of multiple runs of the Petri net: our API supports passing initialization functions that reconfigure the net between runs. However, as there is no standardized way of configuring alternative initial markings or configurations in CPN Tools, the user has to actively make use of this API. In the default configuration, only the immediate net given in the model is evaluated, and no further alternative configurations are explored.

As an example, we show in Listing 3 how we make use of this feature in the MQTT model, where alternative configurations were easily discoverable for us: the signature of MC/DC generation with a simple test-driver is `mcdcgenConfig = fn: int*('a→'b)*'a list*string→unit`, where the first argument is a timeout for the SSE, the second is a function with side effects that manipulates the global configurations that are commonly used in CPN Tools to parameterize models, the next argument is a list of different configurations, followed by the filename for writing results to.

Listing 3 MC/DC tool invocation

```
use(cpnmcdclibpath^"config/simrun.sml");
(* Invocation with default settings
(no timeout) *)
mcdcgen("path/to/mqtt.log");
(* Invocation without timeout; base
model + 2 configurations *)
mcdcgenConfig(0, applyConfig,[co1,co2],
"path/to/mqtt3.log");
```

This function will always first evaluate the initial model configuration and then have additional runs for every configuration. Internally, it calls into CPN Tools' `CalculateOccGraph()` function for the actual SSE. Hence the first `mcdcgen`-invocation in Listing 3 will execute a full SSE without timeout, whereas the second `mcdcgenConfig`-invocation would produce three subsequent runs logged into the same file, again without a default timeout. The test-driver can easily be adapted to different scenarios or ways of reconfiguring a model. Alternatively, traces can also be produced in separate files that are then concatenated for the coverage analysis.

4.3 Coverage visualization in CPN model

To visualize the coverage information in a graphical CPN model, we provide another Python script which parses the CPN model and changes the colour of guards in the CPN model based on coverage data. We take both the original model under test and the coverage results as input arguments and produce a new model where covered arcs and transitions are highlighted in green, whereas the uncovered parts are highlighted in red.

5 Evaluation on example models

In this section, we provide experimental results from an evaluation of our approach to model coverage for CPNs. We present the results of examining eleven (11) non-trivial CPN models from the literature that are freely available as part of scientific publications: a model of the Paxos distributed consensus algorithm [15], a model of the MQTT publish–subscribe protocol [16], three models for distributed constraint satisfaction problem (DisCSP): weak commitment search (WCS), asynchronous backtracking (ABT) and synchronous backtracking (SBT) algorithms [17], a complex model of the runtime environment of an actor-based model (CPNABS) [18], a reactor control system for a nuclear power plant (RCS-NPP) model and Niki T34 Syringe driver model [19]. In addition, we have tested four CPN models for test case generation from natural language requirements (NatCPN) [20]: nuclear power plant (NPP) model, turn indicator system (TIS) model, priority command (PC) model and vending machine (VM) model. All models come with initial markings that allow state space generation, in the case of MQTT, T34PIM and DisCSP complete, and incomplete in the case of Paxos, NatCPN and CPNABS.

5.1 Experimental setup

Figure 2 gives an overview of our experimental setup. Initially, we have the original CPN model under test and we instrument it by transforming SML expressions into a form that as a side effect prints how conditions were evaluated and the overall outcome of the decision (cf. Section 3). Second, we run the SSE on the instrumented model and then reconfigure the configuration (initial marking) with any additional initial configurations if they are obvious from the model. As the side effect of SSE, we run the MC/DC generation which gives as output a log file containing the information of evaluations of conditions in arcs expressions and guards and the decision outcome. Finally, we run the MC/DC analyser (cf. Sect. 4) that determines whether each decision is MC/DC covered or not. In addition, it reports the branch coverage (BC), by checking whether each of the possible branches in each decision has been taken at least once.

Furthermore, we visualize the coverage information in the CPNs models taking as input the original CPN model and the results of how conditions and decisions are MC/DC eval-

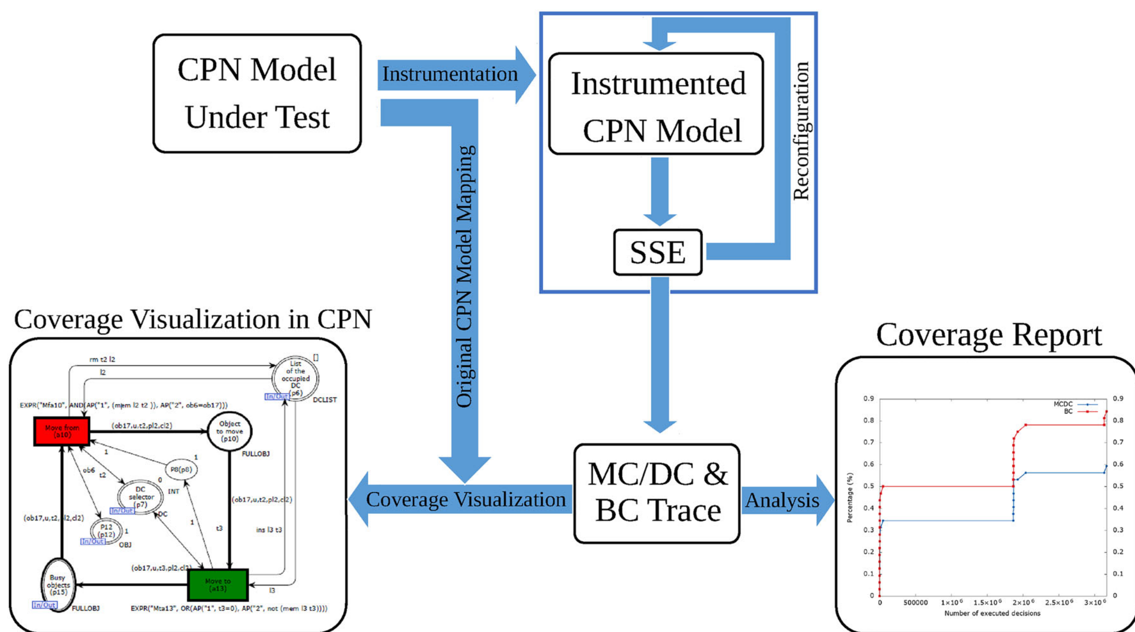


Fig. 2 Experimental setup for Coverage analysis for CPN models

Table 2 MC/DC coverage results for example CPN models

CPN model	Executed decisions	Model decisions	Non- trivial decisions	MC/DC (%)	BC (%)	Simulation status
Paxos	2,281,466	27	11	37.03	40.74	Incomplete
MQTT timeout	3654	18	14	11.11	22.22	Incomplete
MQTT notimeout	1,828,751	23	19	21.73	65.22	Complete
CPNABS	1,386,642	32	13	59.37	88.88	Incomplete
DisCSP WCS	140680	9(2)	5	57.14	57.14	Complete
DisCSP SBT	7686	7	3	57.14	57.14	Complete
DisCSP ABT	604055	7	5	57.14	57.14	Complete
NPP	194,481	13	13	53.84	92.3	Incomplete
PC	8,677,800	10	9	90	90	Incomplete
TIS	10,789,149	19	19	52.94	73.68	Incomplete
VM	4444	8	7	25	50	Incomplete
T34PIM	3,644,768	23	8	69.56	82.6	Complete

uated. This results in the coloured CPN model where the covered parts are coloured in green and the uncovered parts are presented in red. Figure 1a shows a CPN model structure of an original model and Fig. 1b shows the instrumented CPN model after coverage analysis where covered and uncovered parts are highlighted. Table 2 presents the summary of the percentage of how much the tested CPN models are MC/DC and BC covered. The percentage is calculated as the number of covered conditions over the total number of conditions in case of MC/DC and the ratio of covered decisions/branches to the total number of decisions/branches.

5.2 Experimental results

Table 2 presents the experimental results for the eleven example models [15–20]. For each model, we consider the number of executed decisions (second column) in arcs and guards. Column *Model decisions* refers to the number of decisions that have been instrumented in the model. The number of decisions observed in the model and in the log file may deviate in case some of the decisions are never executed, in which case they will not appear in the log file. We indicate them in brackets if during our exploration we did not visit, and

hence log, each decision at least once. In the case of DisCSP, there are two guard decisions which were never executed. The column *Non-trivial decisions* gives the number of the decisions (out of all decisions) that have at least two conditions in the model, as they are the interesting ones while checking independence effect. If a decision has only one condition, it is not possible to differentiate MC/DC from DC. Columns MC/DC(%) and BC(%) present the coverage percentage for the CPN models under test. We record the ratio of covered decisions to the total number of decisions. Due to the large (maybe infinite) state space, we set the timeout to 600 s: in most models, running longer SSE do not increase the coverage metrics in terms of the number of arcs and guards expression executed.

5.3 Discussion of results

MC/DC is covered if all the conditions show the independence effect on the outcome. BC is covered if all the branches are taken at least once. This makes MC/DC a stronger coverage criterion compared to BC, which we will also see in the following graphs. Figure 3 shows the coverage results as the ratio of covered decisions to the number of executed decisions in guards and arcs for both MC/DC and BC. The plots show that the covered decisions increase as the model (and hence the decisions) is being executed. Note that the x-axis does not directly represent execution time of the model: the state space explorer prunes states that have been already visited (which takes time), and hence, as the SSE progresses the number of expressions evaluated per time unit will decrease. In case an expression was executed with the same outcome, the coverage results do not increase, since those test cases have already been explored. Our instrumentation does not have a significant impact on the execution time of the model. Considering the time taken for the full SSE of the finite state models, for instance DisCSP model, both without and with instrumentation, it takes 212 seconds versus 214 seconds, respectively. It is around 1% of overhead which is the cost for the instrumentation.

The CPNABS model and T34PIM model have many single condition (trivial) decisions, and their coverage percentage is higher compared to other models. The Paxos model has less than a half of its decisions covered for both BC and MC/DC with a small percentage difference. The VM model and MQTT with timeout have also less percentage in coverage and both have a high number of non-trivial decisions, which puts more weight on having a suitable test suite to achieve good MC/DC coverage. In addition, we considered additional configurations without timeout for the SSE in the MQTT model and compared the coverage metrics when the configurations are set to timeout. As shown in Fig. 3a, b, the MC/DC and BC percentage increased from 11.11 to 21% and 22.22% to 65.22%, respectively. It is interesting to observe the quality differences of the curves for the tested models. Some of the tested models have less than half of their decisions covered. This should attract the attention of developers and they should assess whether they have tested their models enough, as these results indicate that there is something that might be considered doubtful and require to revisit their test suite. Two factors affect the coverage percentage results presented for these models:

1. The tested models had no clear test suites; they might be lacking test cases to cover the remaining conditions. Depending on the purpose of each model, some of the test cases may not be relevant for the model or the model may not even have been intended for testing. This could be solved by using test case generation for uncovered decisions (see our future work).
2. The models might be erroneous in the sense that some parts (conditions) in the model are never or only partially executed due to a modelling issue, e.g. if the model evolved and a condition no longer serves any purpose or is subsumed by other conditions. For example in the DisCSP model, there are two decisions which were never executed, and we cannot tell whether this was intentionally or not without knowing the goal of the developers.

A main result of our analysis of the example models is that none of the models (including those for which the state space could be fully explored) have full MC/DC or BC. This confirms our hypothesis that code coverage of net inscriptions of CPN models can be of interest to developers, such as revealing not taken branches of the if–then–else arc expressions, never executed guard decisions, conditions that do not independently affect the outcome and some model design errors. Our results show that even for full SSE, we may still find expressions that are not MC/DC covered. Assuming that the model is correct, improving coverage then requires improving the test suite. This confirms the relevance and added value of performing coverage analysis of net inscriptions of CPN models over the dead places/transitions report as part of the state space generation. A natural next step in a model development process would be for the developers to revisit the decisions that are not MC/DC covered and understand the underlying reason. For the models that we have co-published, we can indeed confirm that the original models were not designed with a full test suite in mind, neither from the initial configuration, nor through embedded configurations like for example the MQTT model.

6 Related work

Coverage analysis has attracted attention in both academic and industrial research. In particular, the MC/DC criterion is

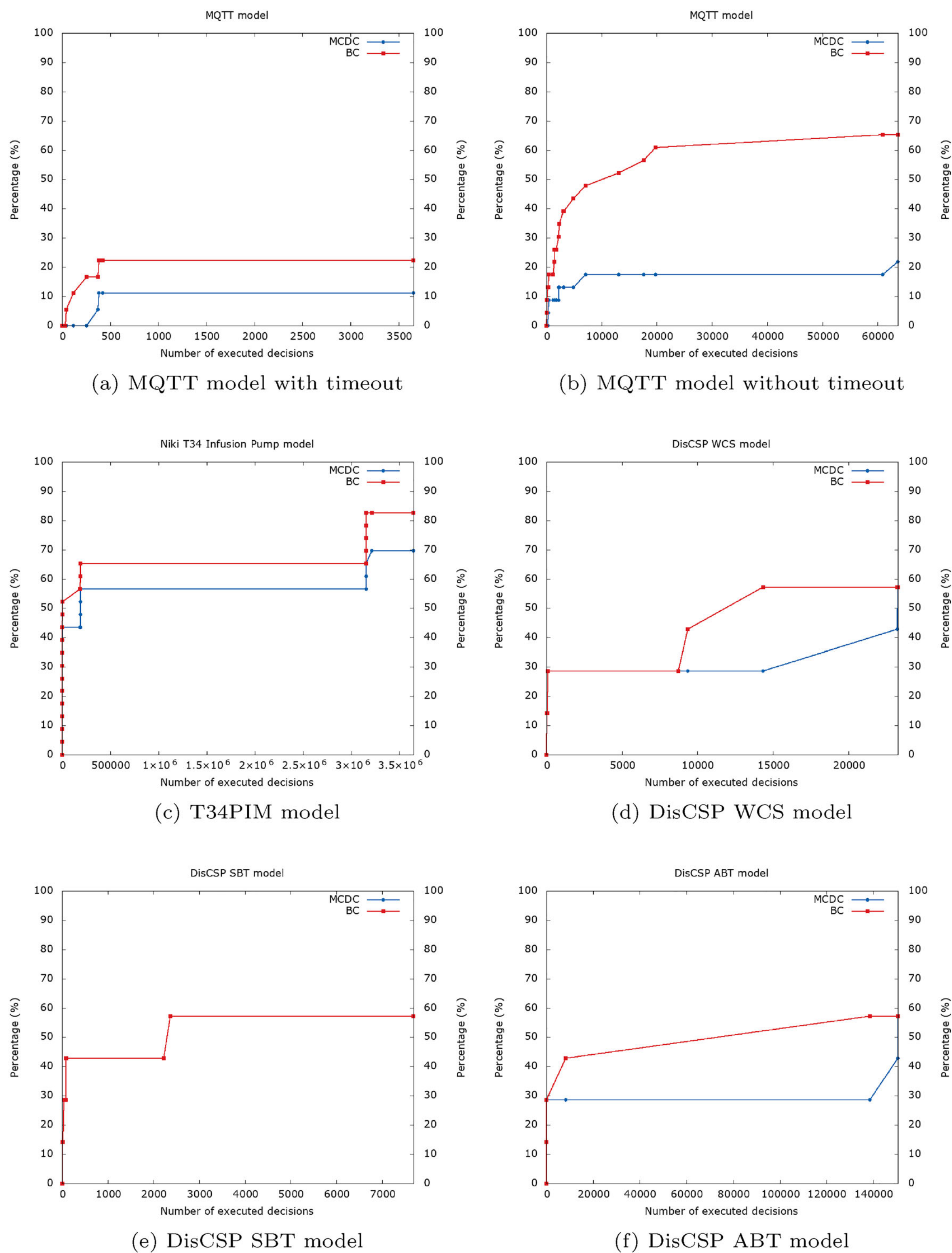


Fig. 3 MC/DC and BC versus number of executed decisions: finite models

highly recommended and commonly used in safety-critical systems, including avionic systems [5]. However, there is a limited number of research addressing model-based coverage analysis. Ghosh [21] expresses test adequacy criteria in terms of model coverage and explicitly lists *condition coverage* and *full predicate coverage criterion* for OCL predicates on UML interaction diagrams, which are semantically related to CPNs in that they express (possible) interactions. Test cases

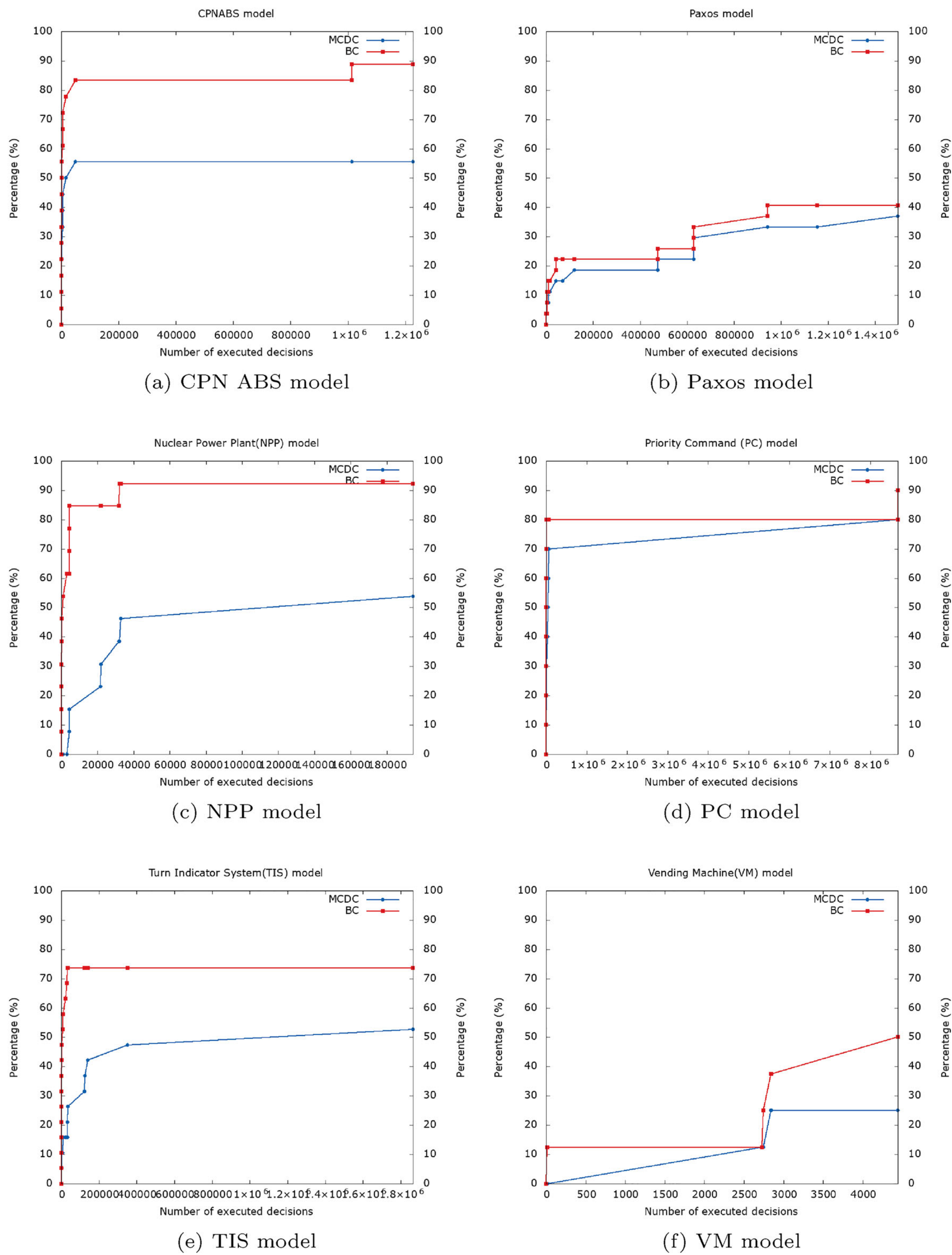


Fig. 4 MC/DC & BC versus number of executed decisions: incomplete models

were not automatically generated. In [22], the authors present an automated test generation technique, MISTA (model-based integration and system test automation) for integrated functional and security testing of software systems using high-level Petri nets as finite state test models. None of the above works addressed structural coverage analysis such as MC/DC or BC on CPN models.

MC/DC is not a new coverage criterion. Chilenski [10] investigated three forms of MC/DC including Unique-Cause (UC) MC/DC, Unique-Cause + Masking MC/DC and Masking MC/DC. Moreover, other forms of MC/DC have been discussed in [23]. More than 70 papers were reviewed and 54 of them discussed MC/DC definitions and the remaining were only focusing on the use of MC/DC in faults detection. We presented in [24], a tool that measures MC/DC based on traces of C programs without instrumentation.

Simulink [25] supports recording and visualizing various coverage criteria including MC/DC from simulations via the Simulink Design Verifier. It also has two options for creating test cases to account for the missing coverage in the design. Test coverage criteria for autonomous mobile systems based on CPNs were presented by Lill et al. in [26]. Their model-based testing approach is based on the use of CPNs to provide a compact and scalable representation of behavioural multiplicity to be covered by an appropriate selection of representative test scenarios fulfilling net-based coverage criteria. Simão et al. [27] provide definitions of structural coverage criteria family for CPNs, named CPN Coverage Criteria Family. These coverage criteria are based on checking whether all markings, all transitions, all bindings and all paths are tested at least once. Our work is different from the above presented work in that we are analysing the coverage of net inscriptions (conditionals in SML decisions) in CPN models based on structure coverage criteria defined by certification standards, such as DO-178C [2].

7 Summary and outlook

We have extended our earlier proof of concept [7] and the supporting software tool to measure MC/DC and branch coverage (BC) of SML decisions in CPN models. There are three main contributions in this paper: (1) We provide a library and automated annotation mechanism that intercept evaluation of Boolean conditions in guards and arcs in SML decisions in CPN models, and record how they were evaluated; (2) we compute the conditions' truth assignments and check whether or not particular decisions are MC/DC covered in the recorded executions of the model; and (3) we collect coverage data using our library from eleven publicly available CPN models and report whether they are MC/DC and BC covered.

We have tested more CPN models and have improved the usability of our instrumentation with respect to the previous release. Firstly, we automate the annotation mechanism that intercepts evaluation of Boolean conditions, which had to be done manually before. We also support the instrumentation of Boolean decision not only in the arcs and guards of CPN models, but also in any SML decision (e.g. function declarations). Secondly, the new release better integrates the coverage analysis tool with the graphical user interface CPN Tools, which supports a broad palette of visual options to indicate successful coverage of guards through colour based on different coverage criteria (MC/DC, BC,...). We leave the encoding of partial functions into delayed evaluation using the so-called *thunks* in SML as future work since it did not pose a problem yet in our example models. Thunks wrap expressions into a constant function that needs to be called to trigger evaluation, and can hence be passed around safely as arguments. As an example, consider `[List.length xs > 0, hd xs]`, which is a valid chain of guards, but will crash in our instrumentation when the list `xs` is empty.

Our experimental results show that our library and post-processing tool can find how conditions were evaluated in all the net inscriptions in CPN models and measure MC/DC and BC. Results reveal that the MC/DC coverage percentage is quite low for the CPN models tested. This is interesting because it indicates that developers may have had different goals when they designed the model, and that the model only reflects a single starting configuration. We can compare this with the coverage of regular software: running a program will yield *some* coverage data, yet most programs will have to be run with many different inputs to achieve adequate coverage.

To the best of our knowledge, our approach is the first work on coverage analysis of CPN models based on BC and MC/DC criteria. This work highlighted that coverage analysis is interesting for CPN models, not only in the context of showing the covered guard and arcs SML decisions, but also the effect of conditionals in SML decisions on the model outcome and related potential problems.

7.1 Outlook

Our general approach to coverage analysis presents several directions forward which would help developers get a better understanding of their models: firstly, while generating the full state space is certainly the preferred approach, this is not feasible if the state space is inherently infinite or too large. Simulation of particular executions could then be guided by results from the coverage and try to achieve higher coverage in parts of the model that have not been explored yet. However, while selecting particular transitions to follow in a simulation is straightforward, manipulating the data space for bindings used in guards is a much harder problem and closely related to test case generation (recall the CPNs also rely on suitable initial states, which are currently given by developers). Making use of feedback between the state of the simulation and the state of the coverage would, however, require much tighter integration of the tools.

As for the measured coverage results, it would be interesting to discuss with the original developers of the models if the coverage is within their expectations. While on the one hand low coverage could indicate design flaws, on the other hand

our testing may not have exercised the same state space as the original developers did: they may have used their model in various configurations, whereof the state of the `git` repository just represents a snapshot, or we did not discover all possible configurations in the model. In the future, we may also try to generate test cases specifically with the aim to increase coverage.

Acknowledgements This work has been partially supported by the Spanish Ministry of Science and Innovation (AwESOMe [PID2021-122215NB-C31]), the Comunidad de Madrid (FORTE-CM [S2018/TCS-4314]) co-funded by EIE Funds of the European Union, EU H2020 project 732016 Continuous Observation of Embedded Multicore Systems COEMS and the SFI Smart Ocean NFR Project 309612/F40.

Funding Open access funding provided by Western Norway University Of Applied Sciences

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Hayhurst KJ, Veerhusen DS, Chilenski JJ, Rierson LK (2001) A practical tutorial on modified condition/decision coverage. Technical Report NASA/TM-2001-210876, NASA Langley Server. https://dl.acm.org/doi/book/10.5555/886632
2. Rierson L (2013) Developing safety-critical software: a practical guide for aviation software and DO-178C compliance, 1st edn. CRC Press, Boca Raton, pp 13–46
3. Jensen K, Kristensen LM (2015) Colored petri nets: a graphical language for formal modeling and validation of concurrent systems. Commun ACM 58:61–70. https://doi.org/10.1145/2663340
4. Jensen K, Christensen S, Kristensen LM, Michael W (2010) CPN tools. http://cpntools.org/
5. Pothon F (2012) DO-178C/ED-12C versus DO-178B/ED-12B: changes and improvements. Technical report, AdaCore
6. John JC, Steven PM (1994) Applicability of modified condition/decision coverage to software testing. Softw Eng J 9(5):193–200
7. Ahishakiye F, Jarabo JR, Kristensen LM, Stolz V (2020) Coverage analysis of net inscriptions in Coloured Petri Net models. In: Hedia BB, Chen Y, Liu G, Yu Z (eds) 14th International conference on verification and evaluation of computer and communication systems (VECOS). LNCS, vol 12519, Springer, Cham, pp 68–83. https://doi.org/10.1007/978-3-030-65955-4_6
8. Stolz V, Jarabo JR, Ahishakiye F, Kristensen LM (2023) Data set for "coverage visualization and analysis of net inscriptions in coloured petri net models" https://doi.org/10.5281/zenodo.7957119
9. Cornett S (1996–2014) Code coverage analysis. Available at https://www.bullseye.com/coverage.html, Accessed 20 Mar 2023
10. John JC (2001) An investigation of three forms of the modified condition decision coverage (MC/DC) criterion. Technical report, Office of Aviation Research
11. Heimdahl MPE, Whalen MW, Rajan A, Staats M (2008) On MC/DC and implementation structure: an empirical study. In: Proceedings of IEEE/AIAA 27th digital avionics systems conference, pp 5–315. https://doi.org/10.1109/DASC.2008.4702848
12. Vilkomir S, Bowen J (2002) Reinforced condition/decision coverage (RC/DC): a new criterion for software testing. In: Proceedings of ZB 2002: formal specification and development in Z and B. LNCS, vol 2272, Springer, Berlin, Heidelberg, pp 291–308. https://doi.org/10.1007/3-540-45648-1_15
13. Certification authorities software team (CAST): rationale for accepting masking MC/DC in certification projects. Technical report, Position Paper CAST-6 (2001)
14. Tofte M (2009) Standard ML language. Scholarpedia 4(2):7515. https://doi.org/10.4249/scholarpedia.7515
15. Wang R, Kristensen LM, Meling H, Stolz V (2019) Automated test case generation for the Paxos single-decree protocol using a Coloured Petri Net model. J Log Algebraic Methods Program 104:254–273. https://doi.org/10.1016/j.jlamp.2019.02.004
16. Rodríguez A, Kristensen L.M, Rutle A (2019) Formal modelling and incremental verification of the MQTT IoT protocol. In: Proceedings of transaction on Petri Nets and other models of concurrency. LNCS, vol 11790, pp 126–145. Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-60651-3_5
17. Pascal C, Panescu D (2017) A Colored Petri Net model for DisCSP algorithms. Concurr Comput Pract Exp 29(18):1–23
18. Gkolfi A, Din CC, Johnsen EB, Kristensen LM, Steffen M, Yu IC (2019) Translating active objects into Colored Petri Nets for communication analysis. Sci Comput Program 181:1–26. https://doi.org/10.1016/j.scico.2019.04.002
19. Caesarea Medical Electronics: Niki T34 syringe pump instruction manual (2008) https://manuals.plus/cme/cme-niki-t34-stringe-pump-manual-pdf
20. Silva BCF, Carvalho G, Sampaio A (2015) Test case generation from natural language requirements using CPN simulation. In: Proceedings of 19th Brazilian symposium on formal methods. LNCS, vol 9526. Springer, Berlin, Heidelberg. pp 178–193. https://doi.org/10.1007/978-3-319-29473-5_11
21. Ghosh S, France R, Braganza C, Kawane N, Andrews A (2003) Orest Pilskalns: test adequacy assessment for UML design model testing. In: Proceedings of 14th international symposium on software reliability engineering, ISSRE'03., pp 332–343. https://doi.org/10.1109/ISSRE.2003.1251054
22. Xu D, Xu W, Kent M, Thomas L, Wang L (2015) An automated test generation technique for software quality assurance. IEEE Reliabil 64(1):247–268. https://doi.org/10.1109/TR.2014.2354172
23. Paul TK, Lau MF (2014) A systematic literature review on modified condition and decision coverage. In: Proceedings of the 29th annual ACM symposium on applied computing. SAC '14, Association for Computing Machinery, New York, USA, pp 1301–1308. https://doi.org/10.1145/2554850.2555004
24. Ahishakiye F, Jakšić S, Stolz V, Lange FD, Schmitz M, Thoma D (2019) Non-intrusive MC/DC measurement based on traces. In: Méry D, Qin S (eds) Intlernational symposium on theoretical aspects of software engineering, IEEE, Guilin, China, pp 86–92 https://doi.org/10.1109/TASE.2019.00-15
25. Simulink: types of model coverage. https://se.mathworks.com/help/slcoverage/ug/types-of-model-coverage.html Accessed 06 Apr 2022
26. Lill R, Saglietti F (2013) Model-based Testing of cooperating robotic systems using Coloured Petri Nets. In: Proceedings of SAFECOMP 2013 - Workshop DECS (ERCIM/EWICS Workshop on Dependable Embedded and Cyber-physical Systems) of the 32nd international conference on computer safety, reliabil-

ity and security, Toulouse, France. https://hal.archives-ouvertes.fr/hal-00848597

27. Simão A, Do S, Souza S, Maldonado J (2003) A family of coverage testing criteria for Coloured Petri Nets. In: Proceedings of 17th Brazilian symposium on software engineering (SBES'2003), pp 209–224

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.