Coverage Games for Testing Nondeterministic Systems

Farn Wang\textsuperscript{1,2}, Jung-Hsuan Wu\textsuperscript{1}, Sven Schewe\textsuperscript{3}, and Chung-Hao Huang\textsuperscript{2}

\textsuperscript{1} Department of Electrical Engineering, National Taiwan University, Taiwan, ROC
\textsuperscript{2} Graduate Institute of Electronic Engineering, National Taiwan University, Taiwan, ROC
\textsuperscript{3} Department of Computer Science, University of Liverpool, UK

Abstract. Modern software systems may exhibit a nondeterministic behavior due to many unpredictable factors. In this work, we propose the node coverage game, a two player turn-based game played on a finite game graph, as a formalization of the problem to test such systems. Each node in the graph represents a functional equivalence class of the software under test (SUT). One player, the tester, wants to maximize the node coverage, measured by the number of nodes visited when exploring the game graphs, while his opponent, the SUT, wants to minimize it. An optimal test would maximize the cover, and it is an interesting problem to find the maximal number of nodes that the tester can guarantee to visit, irrespective of the responses of the SUT. We show that the decision problem of whether the guarantee is less than a given number is NP-complete. Then we present techniques for testing nondeterministic SUTs with existing test suites for deterministic models. Finally, we report our implementation and experiments.

keywords: testing, nondeterminism, coverage, game, strategy, complexity

1 Introduction

Coverage-based techniques \cite{11, 12} have been widely used in the management of testing projects of large and complex software systems. The idea is to model the software under test (SUT) as a finite number of functional equivalence classes (FEC). The number of FECs that a test plan can cover is then used as an indication of completeness of a verification task and quality of the SUT. For white-box testing, typical test criteria include line (statement) coverage, branch coverage, path coverage, dataflow coverage, class coverage, function/method coverage, and state coverage \cite{11, 12}. For black-box testing, popular criteria include input domain coverage, GUI event coverage, etc. According to the empirical study from \cite{7}, coverage-based techniques are effective in detecting software faults.

However, the test coverage of an SUT has to be achieved with various assumptions and nondeterministic responses of the SUT \cite{11, 13}. For example, when we observe how a request message is served by a server SUT, we really have no control whether the server SUT will finish serving the request, deny the request, or be unaware of the request, e.g., due to loss of connection. The best that a verification engineer can do is to use various strategies to try to reach as many FECs of the server SUT as possible. We propose to model this problem as a two player finite-state game \cite{13}, which we call a node coverage game (NC-game for short). The first player is the tester (maximizer; he
for short) and the second is the SUT (minimizer; she for short). The two players play on a finite game graph with nodes for the FECs, where the nodes are partitioned into the nodes that are owned by the tester and the nodes that are owned by the SUT. The tester and the SUT together move a pebble from node to node according to the transition relation of the game graph. The tester chooses the next node when the pebble is on one of his nodes, while the SUT chooses the successor when the pebble is on one of her nodes. The objective of the tester is to maximize the number of visited (covered) nodes, while the SUT wants to minimize the number of covered nodes.

An interesting question about NC-games is how much coverage the tester can guarantee, no matter how the SUT reacts to the test input. We call this guarantee the maximal coverage guarantee (MCG for short). The MCG is calculated under the conservative assumption that the SUT is malicious and tries to minimize the coverage. (For example, a server SUT may decline all interaction requests.) This is common in nondeterministic systems, and a test coverage below the MCG certainly implies deficiency in test execution.

Our two main contributions are as follows.

• We show that the MCG decision problem (the problem whether the SUT can resolve the nondeterminism to prevent a cover over a given threshold \( c \)) is NP-complete. This complexity is lower than those established for related coverage problems [1,9]. (They are PSPACE-complete, cf. Section 2.)

• For a real-world SUT that may exhibit nondeterministic behavior, we propose techniques that combine game concepts and randomness to apply a static test suite developed for deterministic models to test the SUT. We also report our experimental results for these techniques.

In the remainder of the paper, we first review related work in Section 2 and basic concepts of game theory in Section 3. In Section 4, we define NC-games and introduce the MCG decision problem, which we prove to be is NP-complete in Section 5. We then present techniques of applying static test suites to SUT with nondeterminism in Section 6 and report on experimental results using our implementation in Section 7.

2 Related work

A recent work in test coverage games in is the proposition coverage game by Chatterjee, Alfaro, and Majumdar [1]. Their game graph is the same as ours except that each game node is labeled with a set of atomic propositions. The goal of the tester (SUT) is to cover as many (respectively few) propositions as possible. They showed that the decision problem of maximal proposition coverage guarantee is PSPACE-complete.

Another related classic problem is the Canadian traveler problem [9] of Papadimitriou and Yannakakis. A problem instance is a two player game. The players are given a partially observable game graph and do not know which edges are connected. Player 1 may try an edge when he is at the source of the edge. Player 2 then decides whether this edge is connected or not. The connectivity of an edge, once decided, cannot be changed by anyone. The answer to a problem instance is whether the ratio of the traveling distance over the optimal static traveling distance is lower than a given number.
The problem is also PSPACE-complete. In contrast, we show that the MCG decision problem is ‘only’ NP-complete.

In fact, node coverage has been long accepted in the practice of software testing. Thus our result implies that for software testing, coverage analysis can be achieved without incurring PSPACE complexity.

Nachmanson et al. suggested to use random testing to verify nondeterministic SUTs [8]. They argued that, given enough time budget, in general, random testing can lead to good coverage even without prior knowledge of the MCG. In our experiment, we show that, with some game concepts in the test selection, testing nondeterministic SUTs can be made much more efficient.

3 Preliminaries

We first introduce some basic notations. Let \( \mathbb{N} \) be the set of non-negative integers. We write \([i, j]\) for the set of integers inclusively between \( i \) and \( j \). Also, \((i, j), [i, j], \) and \((i, j)\) are shorthands of \([i + 1, j], [i, j - 1], \) and \([i + 1, j - 1]\), respectively.

Given a finite set \( D \), we use \(|D|\) to denote the size of \( D \). Given two sets \( D \) and \( D' \), we use \( D - D' \) to denote the difference set of \( D \) from \( D' \), that is, \( D - D' = \{ d \mid d \in D, d \notin D' \} \).

Suppose that we are given an (infinite) sequence \( \phi = v_0v_1 \ldots \) with elements in a set \( V \). For every \( i \in \mathbb{N} \), we let \( \phi(i) = v_i \). We use \( V^* \) to denote the set of finite sequences of elements in \( V \). Given two sequences \( \phi_1 \) and \( \phi_2 \) such that \( \phi_1 \) is finite, we use \( \phi_1 \phi_2 \) to denote their concatenation.

Our NC-game is played on a directed graph, conventionally called a game graph. Conceptually, a node in the graph represents an FEC of the SUT. An edge represents a transition between FECs. A node can be owned either by the tester (the maximizer, or player 1) or by the SUT (the minimizer, or player 2). At the beginning of the game, there is a pebble in a dedicated initial node of the graph. Depending on who owns the node that contains the pebble, the owner of the node chooses a transition to move the pebble from the current node to the next node. The interleaving of choices by the SUT and the tester extends a play—an infinite sequence of nodes in the game graph. The size of the set of nodes that occur in the play is the coverage of the play.

The above-mentioned concepts can be formalized as follows.

Definition 1. (Game graph) A game graph \( G = (V_1, V_2, E, \nu) \) is a weighted finite directed graph with finite node set \( V_1 \cup V_2 \), edge (transition) set \( E \subseteq (V_1 \cup V_2) \times (V_1 \cup V_2) \), and gain function \( \nu : (V_1 \cup V_2) \to \mathbb{N} \). We require that \( V_1 \cap V_2 = \emptyset \). Nodes in \( V_1 \) are owned by the tester while those in \( V_2 \) are owned by the SUT.

For convenience, from now on, without saying it explicitly, we assume that we are in the context of a given game graph \( G = (V_1, V_2, E, \nu) \). Moreover, we let \( V \) denote \( V_1 \cup V_2 \). Figure 1 shows an example game graph \( G \). Nodes in \( V_1 \) are drawn as circles. Those in \( V_2 \) are drawn as squares. The gain of each node is put down under the node name. A continuous interaction between the SUT and the tester together create an infinite play defined in the following.
Definition 2. (Plays and play prefixes) A play \( \psi \) is a function from \( \mathbb{N} \) to \( V \) such that, for all \( i \geq 0 \), \( (\psi(i), \psi(i + 1)) \) \( \in E \). A play prefix \( \phi \) of \( \psi \) is a mapping from an interval \([0, k]\) to \( V \) such that, for all \( i \in [0, k] \), \( \phi(i) = \psi(i) \).

The following notations are for the convenience of presentation. Given a play prefix \( \phi : [0, k] \rightarrow V \), the length of \( \phi \), denoted \( |\phi| \), is \( k + 1 \). If \( \phi \) is of infinite length, \( |\phi| = \infty \). Given two integers \( j \) and \( k \) in \([0, |\phi|]\) with \( j \leq k \), we use \( \phi[j, k] \) to denote the play prefix \( \phi(j) \cdots \phi(k) \). We use \( \text{last}(\phi) \overset{\Delta}{=} |\phi| - 1 \) to denote the last node in \( \phi \) if the length of \( \phi \) is finite (\( |\phi| \neq \infty \)).

Given a play (or play prefix) \( \psi \), we use \([\psi]\) to denote the domain of \( \psi \), that is, \( [\psi] = \{ \psi(k) \mid k \in [0, |\psi|) \} \). Also, by abuse of notation, \( \nu(\psi) \) denotes the coverage gain of \( \psi \), i.e., \( \nu(\psi) \overset{\Delta}{=} \sum_{v \in [\psi]} \nu(v) \).

Without mentioning it explicitly, we assume that a play has infinite length. A play \( \psi \) with \( \psi(0) = v \) is called a \( v \)-play. In choosing transitions at a node owned by a player, the player may look up the play prefix that leads to the current node, investigate what decisions the opponent has made along the prefix, and select the next node s/he moves to. Such decision-making by a player can be formalized as follows.

Definition 3. (Strategy) A strategy is a function from play prefixes to a successor node. Formally, a strategy \( \sigma \) is a function from \( V^* \) to \( V \) such that for every \( \phi \in V^* \), \( (\text{last}(\phi), \sigma(\phi)) \in E \).

A strategy \( \sigma \) is memoriless (positional) if the choice of \( \sigma \) only relies on the current node of the pebble, that is, for every two play prefixes \( \phi \) and \( \phi' \), \( \text{last}(\phi) = \text{last}(\phi') \) implies \( \sigma(\phi) = \sigma(\phi') \). If \( \sigma \) is not memoriless, it is called memoriful.

Given regular expressions \([6] \varepsilon_1, \ldots, \varepsilon_n \) with alphabet \( V \) and nodes \( v_1, \ldots, v_n \in V \), we may use \( [\varepsilon_1 \rightarrow v_1, \ldots, \varepsilon_n \rightarrow v_n] \) to (partially) specify a strategy. Supposedly, \( \varepsilon_1, \ldots, \varepsilon_n \) should be disjoint from one another. For a strategy \( \sigma \), a rule like \( \varepsilon_i \rightarrow v_i \) means that, for every play prefix \( \phi \in \varepsilon_i \), \( \sigma(\phi) = v_i \). For example, in Figure 1, a memoriless strategy of the tester can be specified with \([V^*v_0 \rightarrow v_1, V^*v_3 \rightarrow v_2]\). A memoriful strategy of the tester can be specified with \([v_0 \rightarrow v_1, v_0V^*v_0 \rightarrow v_2, V^*v_3 \rightarrow v_2]\).

Note that, in Definition 3, we do not distinguish between the strategies of the two players. As a player can only influence the decisions made on his or her nodes, we call a play \( \phi \) \( \sigma \)-conform for a tester strategy \( \sigma \) if, for all \( i \in \mathbb{N} \), \( \phi(i) \in V_i \) implies

\[ \square : \text{owned by the tester; } \circ : \text{owned by the SUT.} \]

Fig. 1. A game graph.
\(\phi(i+1) = \sigma(\phi)\). Likewise, we call it \(\sigma\)-conform for an SUT strategy \(\sigma\) if, for all \(i \in \mathbb{N}\), \(\phi(i) \in V_2\) implies \(\phi(i+1) = \sigma(\phi)\).

In the remainder of the paper, we denote the set of all strategies by \(\Sigma\). Together with an initial node \(r\), two strategies \(\sigma_1, \sigma_2 \in \Sigma\) of the tester and the SUT, respectively, define a unique play, which is conform to both. We denote this play by \(\text{play}(r, \sigma_1, \sigma_2)\).

**Definition 4. (Traps)** For \(p \in \{1, 2\}\), a \(p\)-trap is a subset \(V' \subseteq V\) that player \(3-p\) has a strategy to keep all plays from leaving \(V'\). Formally, we require that:

- For every \(v \in V' \cap V_p\) and every \((v, v') \in E\), \(v' \in V'\).
- For every \(v \in V' - V_p\), there exists a \((v, v') \in E\) with \(v' \in V'\).

For convenience, in this work, a 1-trap is called a tester trap while a 2-trap is called an SUT trap.

## 4 Node coverage game (NC-game)

An NC-game is defined with \(G\) and an initial node.

**Definition 5. (Node coverage game, NC-game)** A node coverage game \(\langle G, r \rangle\) is a pair of game graph \(G\) and an initial node \(r \in V\). In the game, the tester (player 1) tries to cover as many nodes as possible in plays while the SUT (player 2) tries to cover as few nodes as possible in plays.

For convenience, from now on, unless explicitly stated, we assume that we are in the context of an NC-game \(\langle G, r \rangle\). The maximal coverage guarantee (MCG) from \(r\) of \(G\), denoted \(mcg(r)\), is \(\max_{\phi_1 \in \Sigma} \min_{\phi_2 \in \Sigma} \nu(\text{play}(r, \phi_1, \phi_2))\). Intuitively, this is the maximal coverage gain from \(r\) that the tester can guarantee no matter how the SUT may respond.

A strategy \(\phi_1\) of the tester is optimal if it can be used by the tester to achieve at least \(mcg(r)\) coverage no matter how the SUT may respond in the game. Formally, \(\phi_1\) is optimal for the tester if and only if \(\min_{\phi_2 \in \Sigma} \nu(\text{play}(r, \phi_1, \phi_2)) = mcg(r)\) holds. Symmetrically, a strategy \(\phi_2\) of the SUT is optimal if and only if \(\max_{\phi_1 \in \Sigma} \nu(\text{play}(r, \phi_1, \phi_2)) = mcg(r)\) holds.

The complexity of a computation problem in computer sciences is usually studied in the framework of decision problem. Our MCG decision problem is defined as follows.

**Definition 6. (MCG decision problem)** Given a \(c \in \mathbb{N}\), the MCG decision problem asks whether \(mcg(r) \leq c\).

## 5 Complexity of the MCG decision problem

This section contains our main theoretical result: we establish the NP-completeness for the MCG decision problem.

**Theorem 1.** MCG decision problem is NP-complete for both constant and general \(\nu\).

Due to space restrictions, the details of the proofs are moved to the appendix, but we give the main intuition of the proofs below.
5.1 NP-hardness

The hardness is easy to establish by a standard reduction from a SAT problem: we reduce from the satisfiability problem of Boolean formulas in conjunctive normal form (CNF) [3] to the MCG decision problem. As outlined in Figure 2, we translate a Boolean formula $\eta$ in CNF with $n$ atomic propositions $x_1, x_2, \ldots, x_n$ and $m$ clauses $C_1, C_2, \ldots, C_m$ into an NC-game $\langle G, dx_1 \rangle$ as follows.

- We have $m + 3n + 1$ nodes, nodes $x_i$, $\overline{x_i}$, and $dx_i$ for $i = 1, \ldots, n$, a node $y$, and a node $C_j$ for $j = 1, \ldots, m$.
- From the nodes $dx_i$, the SUT can choose to go to $x_i$ or $\overline{x_i}$, respectively.
- From the nodes $x_i$ and $\overline{x_i}$, $dx_{i+1}$ is the only successor for $i < n$, and $y$ is the only successor of $x_n$ and $\overline{x_n}$.
- From $y$, the tester can choose to go to $C_1, \ldots, C_m$.
- For each clause $C_j$ with $1 \leq j \leq m$ and literal $l$ (some $x_i$ or $\overline{x_i}$) in $C_j$, the SUT can go to $l$.

The size of $G$ is $m + 3n + 1$ and the reduction can be done in polynomial time. If the formula is satisfiable, the SUT can force a cover $\leq m + 2n + 1$: intuitively, she can guess a satisfying assignment and restrict the cover to either the node $x_i$ or the node $\overline{x_i}$ for $i = 1, \ldots, n$ (depending on the assignment), $dx_i$ for $i = 1, \ldots, n$, $y$, and $C_j$ for $i = 1, \ldots, m$. If there is no satisfying assignment, she still has to cover $dx_i$ for $i = 1, \ldots, n$ and either the node $x_i$ or the node $\overline{x_i}$ for $i = 1, \ldots, n$ in the first $2n$ steps, and she cannot prevent coverage of all $C_j$ for $i = 1, \ldots, m$. Moreover, when read as an assignment her choice of nodes $x_i$ or the node $\overline{x_i}$ for $i = 1, \ldots, n$ must violate some disjunction $C_j$. When the tester forces the SUT to cover $C_j$, she therefore has to cover one further node $x_i$ or $\overline{x_i}$.

5.2 Inclusion in NP

To show inclusion in NP, we show that the SUT has an optimal strategy that can be described in polynomial space and checked in polynomial time. This may look slightly unusual for the testing community, as a strategy would rather be expected for the tester. But a consequence of our results is that the co-problem of determining if a tester has a strategy of a certain quality is CoNP-complete, and we cannot hope to generally have an optimal tester strategy with a similarly simple description.

We start by outlining how an optimal strategy of the SUT can be described, and then give an intuition on why this is the case.
The key ingredient of our optimal strategy of the SUT is the following. From each node \( v \), the SUT would offer the tester a set \( P_v \) of nodes that the tester can cover. This set is not necessarily a trap since the tester may be in a position to leave it. With the exception of singleton sets \( P_v \), the SUT, however, must have a successor node in \( P_v \) for each of her nodes in \( P_v \).

A sufficient description of an SUT optimal strategy is to provide, for each state \( v \), such a set \( P_v \) and the gain \( c_v \) the tester can at most obtain against the SUT when starting in \( v \). For example, in Figure 3, the SUT must check whether the tester has chosen \( v_1 \) or \( v_2 \) to decide whether to transit to \( v_1 \) or \( v_2 \) from \( v_3 \) to contain the coverage at \( v_3 \) instead of \( v_4 \). Here we can make the following optimal choice:

- \( c_{v_0} = 3 \) and \( P_{v_0} = \{v_0\} \),
- \( c_{v_1} = 2 \) and \( P_{v_1} = \{v_1, v_3\} \),
- \( c_{v_2} = 2 \) and \( P_{v_2} = \{v_2, v_3\} \), and
- \( c_{v_3} = 2 \) and \( P_{v_3} = \{v_2, v_3\} \).

\( c_{v_0} = 3 \) reflects the claim that the SUT can restrict the coverage from \( v_0 \) to three nodes. In order to do so, she first allows the tester to cover \( P_{v_0} = \{v_0\} \). \( P_{v_0} \) is a claim that boils down to “the tester is allowed to gain as much as he can in \( v_0 \) until he decides to leave \( P_{v_0} \) and will not visit \( P_{v_0} \) again.” Calculating \( c_{v_0} \), we find that it is \( |P_{v_0}| + \max\{c_{v_1}, c_{v_2}\} = 3 \).

From \( v_0 \), the tester can choose to either go to \( v_1 \) or to \( v_2 \). This choice determines the next pseudo trap for the tester the system moves to, \( P_{v_1} = \{v_1, v_3\} \) or \( P_{v_2} = \{v_2, v_3\} \), both of which are actually tester traps. The value of \( P_{v_3} \) actually does not matter since the choice at \( v_3 \) is already determined at either \( v_1 \) or \( v_2 \).

Note that Figure 3 also serves as an example, where the SUT needs memory for her optimal decisions. This need for memory is reflected in the strategy above: the choice of the SUT is not made when \( v_3 \) is reached, but when \( v_0 \) is left, either to \( v_1 \) or to \( v_2 \). At this point, the memory is taken into account.

It is easy to see that the consistency of such a guess can be checked in PTIME. Together, the hardness and the inclusion argument provide a proof of Theorem 1. The details of the construction is given in Appendix A.

The proof that an optimal strategy can be described in such a simple way uses the existence of log-consistent strategies as an intermediate lemma. Log-consistency is a weaker requirement: it allows the selection of a successor of an SUT node \( v \) to depend on the history, but if \( v \) itself had occurred before, the same successor needs to
be chosen as before. Given her objective to minimize the cover, the existence of optimal log-consistent strategies for the SUT is not surprising.

Once we restrict our focus on log-consistent strategies, one can view a run as the building of a directed graph. Let us assume we start in a node \( v \) and the SUT follows a log-consistent strategy. Then there is a (not necessarily unique) maximal SCC component that contains \( v \) that can be constructed by the tester. (This might be the trivial component that contains only \( v \) if the tester cannot return to \( v \) at all.)

The \( P_v \) can be chosen to reflect such an SCC component. It is a bit rough in that it only reflects the nodes occurring in this SCC component, but this is enough for the purpose of offering to cover it. Note that, with such an SCC component, the tester could do exactly this: cover it completely and then, leave it in an own node of his choice, unless it is a tester-trap or a trivial SCC component that consists only of a single SUT node.

To close the proof, we have shown that a set \( P_{v'} \) can be chosen after leaving a previously assigned set \( P_v \) to a node \( v' \not\in P_v \) can be selected independent of the history. The details of all these constructions are given in the appendix.

5.3 Taking SUT restart into consideration

From the perspective of a tester, one might argue that it should be possible to re-start tests. A simple reduction from the MCG decision problem with restart to the MCG problem without is also provided in Appendix A. As the hardness argument is not affected, we obtain a similar theorem.

**Theorem 2.** MCG decision problem with re-start is NP complete for both, constant and general \( \nu \).

6 Repetitive test selection with game concepts

There are many academic and commercial tools for generating test suites for deterministic models of SUTs. We here present techniques to apply such test suites to nondeterministic SUTs. Conceptually, a test suite is a set of finite play prefixes (also called test cases) of a game graph. Such test suites can be obtained by viewing the control-flow graph or state-transition diagram of the SUT as a game graph with only one player. For deterministic SUTs, executing each play prefix in the test suite once is enough to get the coverage of the test suite. But for nondeterministic SUTs, this is not the case, since an SUT node in the play prefix may choose a successor different from the one in the play prefix. In traditional settings of software testing, such a diversion from the play prefix is usually interpreted as inconclusive test verdicts or test failure. Thus, straightforward application of such test suites to nondeterministic SUT models may yield low node coverage.

Our idea is to execute those play prefixes, which has been diverted, a few more times in the hope that, when diversion does not always happen, we may cover more nodes. We formalize the idea with the general framework in Algorithm 1. The framework allows us to leverage existing techniques for testing deterministic systems.
Algorithm 1 Repetitive execution of a test suite.

\[
\text{NT-plan}(T, b) \text{ // } T \text{ is a test suite, } b \text{ is the initial budget, } [C] \triangleq \bigcup_{\phi \in C} [\phi]
\]

1: Let \( C = \emptyset \).
2: \textbf{while} \( b > 0 \) and \([C] \neq [T]\) \textbf{do}
3: \hspace{1em} Pick a \( \phi \in T \) with the greatest \( \text{pgain}(\phi, C, T) \).
4: \hspace{1em} Update \( C \) to \( C \cup \{ \phi \} \).
5: \hspace{1em} Execute \( \phi \) and decrement the execution cost from \( b \).
6: \textbf{end while}

Then we can experiment with various testing strategies with different implementations of the function \( \text{pgain}() \). Given a test case set \( C \), we let \([C] \triangleq \bigcup_{\phi \in C} [\phi] \). We also let \( \#C \) be the number of distinct nodes in \( C \), i.e., \( |\bigcup_{\phi \in C} [\phi]| \). In this work, we experiment with the following four strategies. For convenience of discussion, we denote each strategy with a name.

- \textbf{s1.5}: the \textit{deterministic iterative strategy}: Simply execute all \( \phi \) in \( T \) once. This is the deterministic scheme. Specifically, we let \( \text{pgain}(\phi, C, T) \triangleq 1 \) if \( \phi \) has not yet been executed; and \( \text{pgain}(\phi, C, T) \triangleq 0 \) otherwise.
- \textbf{s2}: the \textit{random strategy}: This is to randomly pick test cases that may still lead to more coverage. Specifically, \( \text{pgain}(\phi, C, T) \triangleq 1 \) if \( [\phi] \not\subseteq [C] \); and \( \text{pgain}(\phi, C, T) \triangleq 0 \) otherwise.
- \textbf{s3}: the \textit{random strategy favoring coverage}: This is to randomly pick test cases for more coverage. Formally, \( \text{pgain}(\phi, C, T) \triangleq \text{a random real in } [0, |[\phi] - [C]|] \).
- \textbf{s4}: the \textit{random strategy favoring controlled coverage}: This is to randomly pick test cases that may lead to more coverage with less interference from the SUT. Given a test case \( \phi \), we let \( \alpha(\phi) \) be the number of positions of SUT nodes in \( \phi \), i.e., \( \alpha(\phi) \triangleq |\{i \mid i \in [0, |\phi|), \phi(i) \in V_2\}| \).

\[
\text{pgain}(\phi, C, T) \triangleq \text{a random real in } [0, |[\phi] - [C]|/\alpha(\phi)]
\]

7 Experiments

We have implemented the test plan algorithms introduced in Section 6. We then carried out two experiments to compare their performance with the random walk algorithm [8] and the Ammann and Offutt’s static test plan generator\(^4\) with the node coverage criterion. For the first experiments, we have used four benchmarks from real-world projects. The second experiment uses randomly generated game graphs as benchmarks.

7.1 Benchmarks from real-world projects

We used the following four benchmarks from the web in our experiment. The first three benchmarks are parameterized with \( m \) as a parameter for the concurrency sizes. With parameterized benchmarks, we can collect performance data to see how our techniques scale to the concurrency sizes.

\(^4\) http://cs.gmu.edu/~offutt/softwaretest/
Inventory system [2]: The system consists of a server and several clients. The server maintains the inventory of items. The clients help users to retrieve information pertaining to stored items from the server. In an instance of the benchmark, we have $m$ clients and one server process.

Chat system [4]: The system allows clients to enter chat sessions. Clients that have entered a session may post messages. Each posted message is forwarded to all other clients in the same session. Although the postings are delivered in order, the chat system requires only local consistency. In an instance of the benchmark, we have $m$ clients and up to $m - 1$ pending messages.

Shared multisets [4]: The system allows simultaneous accesses by several threads and supports insertions, deletions, and queries on multisets. The number of threads sharing the multisets is $m$.

Safety Injection System [5]: This is a simplified version of a control system for safety injection that monitors water pressure and injects coolant into the reactor core when the pressure falls below some threshold.

For the inventory system and the safety injection system benchmarks, we manually constructed the extended finite-state machines of the benchmarks and then used REDLIB [10], a model-checking/simulation-checking library, to construct the game graphs. For the chat system and the shared multisets benchmarks, the game graphs are constructed with Spec Explorer [4]. The sizes of the game graphs are given in Table 1.

7.2 Experiment design

We compare six algorithms in our experiment. To be fair, we set a test budget $b$ dollars for all the algorithms. Visiting a node in a test execution consumes one dollar from our budget. Resetting the test execution, i.e., starting a new test case, happens when the SUT does not respond according to the scenarios prescribed in the current test case and costs $d$ dollars. In this experiment, we let $d = 10$ and examine how the algorithms perform with different budget values.

- **rdm** (for random walk): This is the random walk algorithm by [8] that dynamically and randomly explores the game graphs. The random walks stop when the budget runs out.
- **GMU** (for George-Mason Univ., the school of Ammann and Offutt): This is the test plan generator with node coverage criterion by Ammann and Offutt’s software testing tool. The generator accepts a state-transition graph of the SUT and assumes that the SUT is deterministic. The generator then outputs a set of paths from the initial nodes of the graph as the test suite. In our experiment, this algorithm will execute each path in the test suite once. This is very typical of test suite execution for software projects and usually does not consume the whole budget.
- Strategy s1.5, s2, s3, and s4 are respectively executed until the budget runs out. Also, the SUT makes unbiased random choices of the successor nodes. When a random choice does not match the current test case, a reset operation is incurred.

Since randomness is used in the SUT decision and some algorithms, we carried out 100 runs for each benchmark configuration and algorithm and report the average data.
### Table 1. Comparison with the GMU graph coverage tool against industrial benchmarks with random moves

| Benchmark | m nodes | Budget | GMU | s1.5 | s2 | s3 | s4 | rdm |
|-----------|---------|--------|-----|-----|----|----|----|-----|
| inventory system with 1 server and m clients | 20 | 200 | 84.69 ± 0.48% | 82.08 ± 1.11% | 87.19 ± 1.58% | 85.63 ± 1.53% | 84.69 ± 1.41% | 59.26 ± 2.45% |
| | 100 | 800 | 84.69 ± 0.16% | 82.08 ± 1.53% | 87.19 ± 1.58% | 85.63 ± 1.53% | 84.69 ± 1.41% | 59.26 ± 2.45% |
| | 300 | 300 | 84.69 ± 0.16% | 82.08 ± 1.53% | 87.19 ± 1.58% | 85.63 ± 1.53% | 84.69 ± 1.41% | 59.26 ± 2.45% |
| | 600 | 600 | 84.69 ± 0.16% | 82.08 ± 1.53% | 87.19 ± 1.58% | 85.63 ± 1.53% | 84.69 ± 1.41% | 59.26 ± 2.45% |
| | 400 | 400 | 84.69 ± 0.16% | 82.08 ± 1.53% | 87.19 ± 1.58% | 85.63 ± 1.53% | 84.69 ± 1.41% | 59.26 ± 2.45% |
| | 800 | 800 | 84.69 ± 0.16% | 82.08 ± 1.53% | 87.19 ± 1.58% | 85.63 ± 1.53% | 84.69 ± 1.41% | 59.26 ± 2.45% |
| | 1200 | 1200 | 84.69 ± 0.16% | 82.08 ± 1.53% | 87.19 ± 1.58% | 85.63 ± 1.53% | 84.69 ± 1.41% | 59.26 ± 2.45% |
| | 1600 | 1600 | 84.69 ± 0.16% | 82.08 ± 1.53% | 87.19 ± 1.58% | 85.63 ± 1.53% | 84.69 ± 1.41% | 59.26 ± 2.45% |
| | 2000 | 2000 | 84.69 ± 0.16% | 82.08 ± 1.53% | 87.19 ± 1.58% | 85.63 ± 1.53% | 84.69 ± 1.41% | 59.26 ± 2.45% |
| | 5000 | 5000 | 84.69 ± 0.16% | 82.08 ± 1.53% | 87.19 ± 1.58% | 85.63 ± 1.53% | 84.69 ± 1.41% | 59.26 ± 2.45% |
| | 10000 | 10000 | 84.69 ± 0.16% | 82.08 ± 1.53% | 87.19 ± 1.58% | 85.63 ± 1.53% | 84.69 ± 1.41% | 59.26 ± 2.45% |
| chat system with m clients and m−1 pending messages | 65 | 1500 | 84.69 ± 0.16% | 82.08 ± 1.53% | 87.19 ± 1.58% | 85.63 ± 1.53% | 84.69 ± 1.41% | 59.26 ± 2.45% |
| | 1000 | 1000 | 84.69 ± 0.16% | 82.08 ± 1.53% | 87.19 ± 1.58% | 85.63 ± 1.53% | 84.69 ± 1.41% | 59.26 ± 2.45% |
| | 2000 | 2000 | 84.69 ± 0.16% | 82.08 ± 1.53% | 87.19 ± 1.58% | 85.63 ± 1.53% | 84.69 ± 1.41% | 59.26 ± 2.45% |
| | 5000 | 5000 | 84.69 ± 0.16% | 82.08 ± 1.53% | 87.19 ± 1.58% | 85.63 ± 1.53% | 84.69 ± 1.41% | 59.26 ± 2.45% |
| | 10000 | 10000 | 84.69 ± 0.16% | 82.08 ± 1.53% | 87.19 ± 1.58% | 85.63 ± 1.53% | 84.69 ± 1.41% | 59.26 ± 2.45% |
| | 20000 | 20000 | 84.69 ± 0.16% | 82.08 ± 1.53% | 87.19 ± 1.58% | 85.63 ± 1.53% | 84.69 ± 1.41% | 59.26 ± 2.45% |
| | 50000 | 50000 | 84.69 ± 0.16% | 82.08 ± 1.53% | 87.19 ± 1.58% | 85.63 ± 1.53% | 84.69 ± 1.41% | 59.26 ± 2.45% |
| | 100000 | 100000 | 84.69 ± 0.16% | 82.08 ± 1.53% | 87.19 ± 1.58% | 85.63 ± 1.53% | 84.69 ± 1.41% | 59.26 ± 2.45% |
| | 200000 | 200000 | 84.69 ± 0.16% | 82.08 ± 1.53% | 87.19 ± 1.58% | 85.63 ± 1.53% | 84.69 ± 1.41% | 59.26 ± 2.45% |
| | 500000 | 500000 | 84.69 ± 0.16% | 82.08 ± 1.53% | 87.19 ± 1.58% | 85.63 ± 1.53% | 84.69 ± 1.41% | 59.26 ± 2.45% |
| | 1000000 | 1000000 | 84.69 ± 0.16% | 82.08 ± 1.53% | 87.19 ± 1.58% | 85.63 ± 1.53% | 84.69 ± 1.41% | 59.26 ± 2.45% |

#### 7.3 For the industrial SUTs

The performance data of the first experiment is in Table 1. We measured the average coverage gains achieved with different budget values. We have also drawn the charts with respect to the table for each benchmark in Figures 4 and 5. In general, our algorithm s2, s3, and s4 usually increase coverage much faster than the others. The only exception is for the chat system with m = 6. For this benchmark, the random walk strategy performs best. When we examine the game graph, we found that there are many SUT nodes with many successors. Thus, the test cases from a static test suite can easily run into failure and need reset. In comparison, the random walk algorithm can always accept the choice of the SUT and continue the walk.

#### 7.4 For randomly generated game graphs

To avoid our bias in selecting the benchmarks, we also experimented with randomly generated game graphs. We randomly generated 8 game graphs. The performance data is in Table 2. We have also drawn the charts with respect to the table for each benchmark in Figure 6. The experiment data matches our observation in the last experiment, that is,
Fig. 4. Performance Data of Industrial Benchmarks

algorithm s2, s3, and s4 usually increase coverage much faster than the others. This may imply that testing a nondeterministic system with game concepts could be promising in improving the verification performance.
8 Conclusion

We propose to design test plans for covering software products with game concepts. We have established that the MCG decision problem is NP-complete while previous frameworks for game graph coverage [1, 9] are PSPACE-complete. This may imply that our framework of NC-games can bring about more efficiency in various computing aspects of software testing than those of [1, 9]. We have also presented techniques to leverage on existing test plan generators for testing SUTs with nondeterminism. The experimental results are promising and demonstrate the potential of our game-based techniques. We feel that our work not only have laid a concrete foundation for further investigation in this regard, but could also be useful in testing real-world interactive and embedded software systems.

References

1. K. Chatterjee, L. Alfaro, and R. Majumdar. The complexity of coverage. In _6th Asian Symposium on Programming Languages and Systems (APLAS)_ , volume LNCS 5356, pages 91–106. Springer-Verlag, 2008.
2. CONFORMIQ. Company website: http://www.conformiq.com/.
3. S. Cook. The complexity of theorem proving procedures. In _3rd ACM Symposium on Theory of Computing (STOC)_ , pages 151–158. Association for Computing Machinery (ACM), 1971.
Fig. 6. Performance Data of Random Game Graphs
Table 2. Comparison of the GMU graph coverage tool against random game graphs with random moves

| Game graphs | Nodes | Budget | GMU | s1.5 | s2 | s3 | s4 | rdm |
|-------------|-------|--------|-----|------|----|----|----|-----|
| G1          | 19    | 51.7±0.9% | 64.33±2.52% | 60.4±2.65% | 76.37±2.18% | 70.29±2.33% | 11.53±2.4% |
| G2          | 30    | 97.06±1.35% | 98.16±1.26% | 99.14±0.59% | 97.08±1.35% | 93.33±3.00% |
| G3          | 40    | 78.26±0.75% | 78.52±0.68% | 74.11±1.12% | 91.31±1.52% | 87.29±1.15% |
| G4          | 46    | 97.01±1.19% | 98.55±0.52% | 98.46±0.56% | 97.44±0.69% | 96.63±1.28% |
| G5          | 75    | 89.02±0.66% | 93.06±0.71% | 95.7±0.62% | 94.33±0.64% | 92.76±1.26% |
| G6          | 95    | 98.34±0.58% | 98.63±0.39% | 98.07±0.32% | 94.23±0.89% | 92.95±0.47% |
| G7          | 100   | 99.02±0.58% | 99.44±0.84% | 99.39±0.57% | 92.2±0.41% | 92.97±0.28% |
| G8          | 297   | 84.37±0.47% | 86.9±0.64% | 84.26±0.62% | 85.03±0.25% | 84.7±0.43% |

4. S. Explorer. Tool webpage: http://research.microsoft.com/specexplorer, January 2005.
5. G. Fraser and F. Wotawa. Test-case generation and coverage analysis for nondeterministic systems using model-checkers. In 2nd International Conference on Software Engineering Advances, pages 45–50. IEEE Computer Society, 2007.
6. J. Hopcroft and J. Ullman. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, 1979.
7. M. R. Lyu, Z. Huang, S. K. S. Sze, and X. Cai. An empirical study on testing and fault tolerance for software reliability engineering. In International Symposium on Software Reliability Engineering (ISSRE), page 119, Los Alamitos, CA, USA, 2009. IEEE Computer Society.
8. L. Nachmanson, M. Veanes, W. Schulte, N. Tillmann, and W. Grieskamp. Optimal strategies for testing nondeterministic systems. In ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), pages 55–64, New York, NY, USA, 2004. ACM.
9. C. H. Papadimitriou and M. Yannakakis. Shortest paths without a map. Theoretical Computer Science, 84:127–150, 1991.
10. F. Wang. Redlib for the formal verification of embedded systems. In Leveraging Applications of Formal Methods, Second International Symposium, ISoLA 2006, Paphos, Cyprus, 15-19 November 2006, pages 341–346. IEEE, 2006.
11. E. J. Weyuker. How to judge testing progress. Journal of Information and Software Technology, 45(5), 2004.
12. E. J. Weyuker. In defense of coverage criteria. In Proceedings of the 11th ACM/IEEE International Conf. on Software Engineering(ICSE), May, 1989.
13. M. Yannakakis. Testing, optimization, and games. In IEEE LICS, pages 78–88, 2004.
APPENDICES

A Full version of the Complexity of MCG

We start and end this section with the complexity of the MCG decision problem. It turns out that establishing the lower bound of the MCG decision problem is much simpler than establishing a matching upper bound. We first prove a lemma that comes in handy for establishing the upper bound in Subsection A.2. In the following lemma, we show that the optimal node coverage problem is NP hard.

A.1 NP-hardness

Note, however, that we show the existence of an SUT strategy; consequently testing the existence of an SUT strategy is CoNP hard.

Lemma 1. The MCG decision problem is NP-hard, even for constant $\nu$.

Proof. We reduce the satisfiability problem of Boolean formulas in conjunctive normal form (CNF) [3] to the MCG decision problem. Suppose we have a CNF formula with atomic propositions $x_1, x_2, \ldots, x_n$ and clauses $C_1, C_2, \ldots, C_m$. We construct a node coverage game $(G, dx_1)$ as follows.

- $V_1 = \{y\} \cup \{\bar{x}_i, x_i \mid 1 \leq i \leq n\}$. $\bar{x}_i$ resp. $x_i$ represent, for each atomic proposition $x_i$, a node that interprets $x_i$ as false resp. true. $y$ is a special node intuitively used to force the truth valuation of an arbitrary clause (and, ultimately, of all clauses).
- $V_2 = \{dx_i \mid 1 \leq i \leq n\} \cup \{C_1, \ldots, C_m\}$. Each node $dx_i$ represents a decision node to choose between interpretations $x_i = \text{false}$ and $x_i = \text{true}$. Each node $C_j$ is used for the truth valuation of clause $C_j$.
- $E$ is a minimum set satisfying the following restrictions.
  - For each $1 \leq i \leq n$, $(dx_i, \bar{x}_i) \in E$ and $(dx_i, x_i) \in E$.
  - For each $1 \leq i < n$, $(\bar{x}_i, dx_{i+1}) \in E$ and $(x_i, dx_{i+1}) \in E$.
  - $(\bar{x}_n, y) \in E$ and $(x_n, y) \in E$.
  - For each $1 \leq j \leq m$, $(y, C_j) \in E$.
  - For each clause $C_j$ with $1 \leq j \leq m$ and literal $l$ in $C_j$, $(C_j, l) \in E$.
- $dx_1$ is the starting node of the game.
- For each node $v$ in $V$, $\nu(v) = 1$.

For example, in Figure 2, we have the game graph for a Boolean formula. The size of $G$ is $m + 3n + 1$ and the reduction can be done in polynomial time.

It now suffices to show that a CNF formula $\Psi$ is satisfiable if, and only if, the optimal node coverage with the game graph described is above is $m + 2n + 1$ or smaller (it is precisely $m + 2n + 1$, but this is unimportant for the proof), and unsatisfiable if, and only if, it is strictly bigger.

We prove this claim as follows in two directions. Let us first assume that the minimal coverage is $m + 2n + 1$ (or less). We observe that the minimal coverage must always include all $dx_i$ ($n$ nodes) and $y$ (1 node), because these nodes are always among the first $2n + 1$ visited nodes, irrespective of the chosen strategies, and all $C_j$ ($m$ nodes),
because the tester can reach each of these nodes from any position of the game. Further, we observe that, within the first 2n visited nodes, there is, for all i ∈ {1, . . . , n}, either a visit to node xi or to node x3i. These are already m + 2n + 1 nodes.

The set of nodes xi or x3i visited within the first 2n steps is fixed at the beginning. If the minimal coverage shall be (less than or) equal to m + 2n + 1, then the tester may not be able to cover any additional node. But as the tester can reach each C j, this means that, for each C j, some successor node (which is either an xi or an x3i node) must have been covered in the first 2n steps. According to the construction of the game, the interpretation I from {x1, . . . , xn} to {true, false} that maps each xi to true if xi is covered (and to false if x3i is covered) makes the CNF formula Ψ true.

Vice versa, let us fix such an interpretation I. We now consider a strategy of the SUT that moves from dx1 to xj if xj is evaluated to true by I and to x3i if x3i is evaluated to false by I, and to turn from each C j to a node xj (resp. x3i) such that the respective literal xj (resp. ¬x3i) in Cl is true. Clearly, the coverage is m + 2n + 1.

A.2 Log-Consistent strategies

The main theorem in this appendix is the NP completeness of the MCG decision problem. We then continue with the observation that the complexity of the related problem of testing with re-start (that is, with the capability to re-start the test arbitrarily many times) results in the same complexity.

Before we turn to the NP completeness proof, we will establish that the SUT can play log-consistent, that is, it can make the same decision everytime it comes to the same node. Note that this is different to memoriless, as the history at the first time it visits v may determine the choice.

For example, in Figure 3, the optimal strategy for the SUT is log-consistent but not memoriless. The SUT must check whether the tester has chosen v1 or v2 to decide whether to transit to v1 or v2 from v3 to contain the coverage at 3 instead of 4. Log-consistent strategies make sense for the SUT since choosing an already-executed transition does not increase the coverage gain for the tester. Log-consistent strategies prove to be sufficient for optimal strategies of the SUT.

The formal definition of log-consistent strategies follows.

Definition 7. (Log-consistent strategies) A strategy σ ∈ Σ of the SUT is log-consistent if, for every σ-conform play φ and all k, l ∈ N, it holds that φ(k) = φ(l) ∈ V2 it holds that φ(k + 1) = φ(l + 1).

For example, with the game graph from Figure 3, a strategy satisfying [v0v1(v0v1)v3] v1  v1, v0v2(v3v2)v3  v2] is log-consistent while a strategy satisfying [v0v3v1v3] v2, v0v3v2v3  v1] is not log-consistent.

Lemma 2. In an NC-game, the SUT has an optimal strategy, which is log-consistent.

Proof. Assume that this is not the case. Let σ−1 be an optimal strategy in that it guarantees minimal gain e for the SUT, which may or may not be log consistent. We define a sequence of SUT strategies σ0, σ1, σ2, σ3, . . . as follows.

Let φ = v0v1v2 . . . vi be the prefix of a σi-1-conform play. If vi ∈ V2, then we choose a φ′ such that
1. \( \phi \phi' \) is the prefix of a \( \sigma_{i-1} \)-conform play,
2. \( \phi \phi' \) ends in \( v_i \), and
3. among such \( \phi \phi' \), the states covered is maximal; that is, irrespective of the strategy of the tester, no further state is covered, on any finite sequence that returns to \( v_i \).

We use this \( \phi' \) to infer \( \sigma_i \) from \( \sigma_{i-1} \) as follows. We define for every history \( \phi \phi_1 \phi_2 \) with last(\( \phi \phi_1 \)) = \( v_i \) and \( v_i \notin \llbracket \psi \rrbracket \) (that is, \( \phi_1 \) is either empty or ends in \( v_i \) and \( \phi_2 \) does not contain \( v_i \)), \( \sigma_i(\phi \phi_1 \phi_2) = \sigma_{i-1}(\phi \phi_2) \).

We now select the limit strategy \( \sigma_{\infty} = \lim_{n \to \infty} \sigma_i \). \( \sigma_{\infty} \) is well defined, as the reaction on a play prefix of length \( i \) is the same as the reaction of \( \sigma_i \). The way we update the function clearly ensures log-consistency. We show optimality by induction.

As induction basis, \( \sigma_{-1} \) is optimal by assumption.

For the induction step, let us assume for contradiction that \( \sigma_{i-1} \) is optimal, but \( \sigma_i \) is not. Let us assume for contradiction that \( \psi = v_0 v_1 \ldots v_i \ldots \) is a \( \sigma_i \)-conform play with \( \nu(\psi) > c \). We now distinguish two cases.

1. \( v_i \) occurs infinitely often in \( \psi \). But then, by condition (3) of our construction, the nodes covered by \( \psi \) are covered by the play prefix \( \phi \phi' \) of a \( \sigma_{i-1} \)-conform play. Consequently, \( \llbracket \psi \rrbracket \subseteq \llbracket \phi \phi' \rrbracket \) holds, which implies \( \nu(\psi) \leq \nu(\phi \phi') \). As our induction hypothesis in particular implies \( \nu(\phi \phi') \leq c \), this contradicts the assumption \( \nu(\psi) > c \).
2. \( v_i \) occurs only finitely often, say, it occurs last at position \( k \geq i \). Let \( \psi' = v_0 v_1 \ldots v_k \) and \( \psi = \psi' \psi'' \). Then, by the same argument as above, the states covered in \( \psi' \) are contained in the states covered by \( \phi \phi' \) (\( \llbracket \psi' \rrbracket \subseteq \llbracket \phi \phi' \rrbracket \)). Also, by construction, \( \phi \phi' \psi'' \) is a \( \sigma_{i-1} \)-conform play. We thus obtain \( \llbracket \psi' \psi'' \rrbracket \subseteq \llbracket \phi \phi' \psi'' \rrbracket \), and consequently \( \nu(\psi') \leq \nu(\phi \phi' \psi'') \). As our induction hypothesis provides \( \nu(\phi \phi' \psi'') \leq c \), this contradicts the assumption \( \nu(\psi') > c \).

After having established that all \( \sigma_i \) are optimal, let us assume for contradiction that \( \sigma_{\infty} \) is not, that is, that there is a \( \sigma_{\infty} \)-conform play \( \phi \) with \( \nu(\phi) = c' > c \). But then, some finite initial sequence of \( \phi' \) of \( \phi \) must have a gain \( c' = \nu(\phi') \), and \( \phi' \) must be a prefix of a \( \sigma_{|\phi'|} \) conform play \( \psi \). Consequently \( \nu(\psi) \geq \nu(\phi') > c \) holds (contradiction).

A.3 Inclusion in NP

Having established that it is enough to consider log-consistent strategies, we outline an algorithm that guesses a witness SUT strategy that includes, for each node \( v \) of the game,

- the coverage of \( c_v \) that the SUT allows the tester to obtain if we start from this node (that is, on a game that is modified only in that the initial node is changed to \( v \)), and
- a set \( P_v \subseteq V \) (for pseudo trap from \( v \)) of nodes that includes \( v \) and satisfies the following constraints:
  - \( P_v = \{ v \} \) is singleton and \( v \) is an SUT node, or
  - for SUT node \( u \) in \( P_v \), there is a node \( w \) in \( P_v \) such that \( (u, w) \in E \).

iii
The game is then intuitively divided into two phases: the ‘covering phase’, where \( P_v \) may be covered by the tester, followed by a phase where the game would not return to \( P_v \).

We will argue that the SUT can play optimal by playing from a node \( v \) reached after \( P_v \) is left as if the game would start in \( v \). Not returning to \( P_v \) would merely be a property of an optimal SUT strategy that starts in \( v \) and not technically required.

A strategy of the SUT is called a witness strategy if the SUT can follow the strategy
- to stay in \( P_v \) by playing, from a node \( u \in P_v \), an edge \((u, u')\) with \( u' \in P_v \), and
- to follow the strategy for starting in \( v \) as soon as \( P_v \) is left to \( w \).

The SUT may use this witness strategy to trap plays in \( P_v \) until the tester decides that he cannot gain more coverage in \( P_v \) and had better leave \( P_v \) for \( w \).

A witness is consistent if it satisfies the following side constraints:

- **Case S** (for singleton): if \( P_v = \{v\} \), \( v \) is an SUT node, \( v \) is no sink, and the self-loop \((v, v)\) is not an edge of the game, then we have \( c_v = \min\{c_u+1 \mid (v, u) \in E\} \), and

- **Case T** (for trapping): otherwise, let \( c_v = \max\{c_w \mid u \in P_v \cap V_1 , (u, w) \in E, w \notin P_v \} + |P_v| \), where the maximum over the empty set is 0 (to account for the case that \( P_v \) is a tester trap).

Consistency is easy to check.

**Lemma 3.** Consistency can be checked in polynomial time.

To follow a consistent witness, the SUT needs a set variable \( T \) to record the current set of nodes that the tester is allowed to cover. The following steps describe how a consistent witness strategy can be followed. Initially, \( T = \emptyset \). At a node \( v \), the SUT first updates \( T \) according to the following cases.

- If \( v \) is a case S node, \( T \) is set to \( \emptyset \).
- If \( v \) is a case \( T \) node, the following cases are further considered.
  - If \( T = \emptyset \), then \( T \) is set to \( P_v \). (This refers to the case that \( v \) is the initial node.)
  - If \( v \in T \), then we are still in the same set of nodes that the witness strategy has allowed the tester to cover. Thus, we ignore \( P_v \) and let \( T \) stay unchanged.
  - If \( v \notin T \), then we just left the last set of nodes that the witness strategy allowed the tester to cover. Thus, we reset \( T \) to \( P_v \). Note that \( v \) may be a tester node.

Then at a node \( v \in V_2 \), the SUT makes the following decision.
- If \( T = \emptyset \), this implies that \( v \) is a case S node and the SUT should pick a successor \( u \) with \((v, u) \in E\) and \( c_u = \min\{c_w \mid (v, w) \in E\} \).
- If \( T \neq \emptyset \), the SUT should choose a \( u \in T \) with \((v, u) \in E\).

**Lemma 4.** If \( \nu(v) \geq 1 \) for all \( v \in V \) then the SUT can, for every consistent witness, guarantee a gain of at most \( c_v \) from every node \( v \in V \).

**Proof.** We show this by induction over \( c_v \). For the induction basis, this is clearly true if \( P_v \) is a tester trap, as this implies that \( \{c_w \mid u \in P_v \cap V_1 , (u, w) \in E, w \notin P_v \} \) is empty, and consequently \( c_v = \nu(P_v) \) holds. Note that this is in particular the case for \( c_v = 1 \).
For the induction step, we can follow the witness strategy to obtain all guarantees smaller than \( c_v \) by induction hypothesis. We now distinguish two cases.

1. **Case S** of \( v \): Then there must be a successor \( u \) (with \( (v, u) \in E \)) with \( c_u = c_v - \nu(v) < c_v \). By induction hypothesis, the SUT can restrict the gain to \( c_u \) using the witness strategy, and consequently to \( c_v \) from \( v \).

2. **Case T** of \( v \): Following a witness strategy allows to cover first (some or all) nodes in \( P_v \), and then might continue to a successor \( w \) of any tester node in \( P_v \). (Recall that a witness strategy of the SUT will not leave \( P_v \) from any SUT node.) We have \( c_v > \max \{ c_x \mid u \in P_v \cap V_1, (u, x) \in E, x \notin P_v \} \geq c_w \) by the consistency of the witness. Thus, the tester can guarantee a gain of at most \( c_w \) after moving to \( w \), while clearly at most \( \nu(P_v) \) has been gained before. ■

Note that the correctness argument does not require that \( P_v \) is not visited again. It is indeed possible to construct consistent witnesses who do this – not all consistent witnesses are optimal. We now show that an optimal consistent witness exists.

**Theorem 3.** For a game with \( \nu(v) \geq 1 \) for all \( v \in V \) where an optimal SUT can restrict the gain to \( m_v \) when starting in node \( v \), there is a consistent witness with \( c_v = m_v \) for all nodes \( v \).

**Proof.** We show by induction that, for games with a minimal gain of \( m_v \), we can construct a witness with \( c_v = m_v \) for all \( v \in V \). By the previous lemma, this shows that the SUT can force that, when starting at \( v \), the gain is bounded by \( m_v \).

For the induction basis, this is clearly true if there is a tester trap \( P_v \) with \( m_v = \nu(P_v) \). For such tester traps, we can simply select \( c_v = \nu(P_v) = m_v \). Note that this in particular includes the case \( m_v = 1 \).

For the induction step, we exploit that log-consistent strategies are sufficient for the SUT (Lemma 2). Let us assume that \( m_v \) is the correct value, and that the SUT uses a fixed log-consistent optimal strategy. Then, we can look at a run as a producer of edges in a directed graph: we start with the game intersected with the transitions whose source is a tester node. That is, we remove exactly the transitions that exit an SUT node.

During the run, every time we pass by an SUT node, we add the selected transition to the di-graph. For a run, we choose the maximal of these graphs such that the added transition belongs to the same SCC as the starting node \( v \).

For every run, there is obviously one such graph, and there is obviously a (not necessarily unique) graph among them where this SCC is maximal. We select such a graph \( G \) and select \( P_v \) to be the set of nodes in the same SCC as \( v \). For the SUT nodes in this SCC, we memorize the (due to log-consistency unique) outgoing edge selected by the strategy.

Moreover, we pick, for each successor \( w \) of a tester node in \( P_v \), a history \( h_w \) consistent with our log-consistent optimal strategy such that (1) initially \( P_v \) is covered completely and (2) then a transition to \( w \) is taken. It is obvious that such a history exists, as the play in which \( P_v \) is first an SCC can be extended to cover \( P_v \) and then to move on to \( w \). From \( w \), the SUT will play as if it used the old strategy from \( v \) and had previously seen \( h_w \). We call the inferred strategy of the SUT to continue from \( w \)
Note that a \( w \)-consistent strategy is (1) log-consistent and (2) guarantees that \( P_v \) is not visited again, as \( P_v \) is a maximal SCC.

Note that the following strategy provides the same guarantees: while in \( P_v \), stay in \( P_v \), using the transitions from \( G \); once \( P_v \) is left to a node \( w \), follow the \( w \)-consistent strategy. It is therefore an optimal log-consistent strategy from \( v \). (Optimal because the SUT has the described strategy to achieve the same guarantees as under the optimal log-consistent strategy we started with.)

Note that, by construction, \( P_v \) is not reachable from \( w \) under the \( w \)-consistent strategy. (If it was, the SCC was not maximal.) Consequently, we have \( c_w < c_v \), and, by induction hypothesis, we verify the correct \( c_w \) using our witnesses.

Let us assume for contradiction that this is smaller than the result using the \( w \)-consistent strategy from \( w \). Then the strategy from above can be strictly improved to: while in \( P_v \), stay in \( P_v \), using the transitions from \( G \), once \( P_v \) is left to \( w \), follow an optimal strategy that starts in \( w \). This provides a contradiction to the optimality of this strategy.

This leaves the special case where \( P_v = \{v\} \) contains a single SUT node without a self-loop, but with some successor. In this case, we have a clearly defined \( w \)-consistent strategy for the successor \( w \) selected by the SUT (and memorize the edge \((v, w)\), together with the guarantee that, in the \( w \)-consistent strategy, \( v \) cannot be visited. This provides us with a strategy for the SUT to restrict the gain from \( w \) to \( c_w - \nu(v) \).

Note that the restriction to \( \nu(v) \geq 1 \) is no restriction. For a game with \( n \) nodes, we can use the function \( \nu' \) with \( \nu'(v) = 1 + n \cdot \nu(v) \) instead: a strategy is optimal for \( \nu \) if it is optimal for \( \nu' \) (though the converse does not hold).

Putting the lemmata of this section together, we obtain inclusion in NP with this observation. With the matching hardness result from Lemma 1, we get our main theorem.

\[ \text{Theorem 1. MCG decision problem is NP complete for both, constant and general } \nu. \]

A slight variation of the problem is to allow the tester to re-start the game at any time. This variation is natural as one can argue that a test can be re-started.

We first note that the hardness proof is not affected: the SUT can repeatedly guess the same satisfying assignment for the CNF SAT problem.

For the inclusion, we can consider two games: the game with re-set, and a game where every node is doubled into an in- and an out-node, both with the same gain. A transition from \( v \) to \( w \) becomes a transitions from the out-node of \( v \) to the in-node of \( w \), and the new initial node is the in-node to the old initial node. The in-node of \( v \) has two outgoing transitions: one to the initial node and one to the out-node of \( v \). While all in-nodes are tester nodes, the out-node of \( v \) belongs to the player that owned \( v \).

The only other change applied is that, if \( v \) is a sink in the game with re-set, then we add a transition from the out-node of \( v \) to the new initial node.

It is now easy to show that the maximal gain of the re-start game is exactly half the maximal gain of the new game. We can show this by simulating the games.

\[ \text{Theorem 2. MCG decision problem with re-start is NP complete for both, constant and general } \nu. \]