The Cubicle Fuzzy Loop : A Fuzzing-Based Extension for the Cubicle Model Checker

Sylvain Conchon, Alexandrina Korneva

The Cubicle Fuzzy Loop : A Fuzzing-Based Extension for the Cubicle Model Checker

Sylvain Conchon and Alexandrina Korneva

Université Paris-Saclay, CNRS, ENS Paris-Saclay, Laboratoire Méthodes Formelles, 91190, Gif-sur-Yvette, France.

Abstract. This paper presents the Cubicle Fuzzy Loop (CFL), a fuzzing-based extension for Cubicle, a model checker for parameterized systems. To prove safety, Cubicle generates invariants, making use of forward exploration strategies like BFS or DFS on finite model instances. However, these standard algorithms are quickly faced with the state explosion problem due to Cubicle's purely nondeterministic semantics. This causes them to struggle at discovering critical states, hindering invariant generation.
CFL replaces this approach with a powerful DFS-like algorithm inspired by fuzzing. Cubicle's purely nondeterministic execution loop is modified to provide feedback on newly discovered states and visited transitions. This feedback is used by CFL to construct schedulers that guide the model exploration. Not only does this provide Cubicle with a bigger variety of states for generating invariants, it also quickly identifies unsafe models. As a bonus, it adds testing capabilities to Cubicle, such as the ability to detect deadlocks.
Our first experiments have yielded promising results. CFL effectively allows Cubicle to generate crucial invariants, useful to handle hierarchical systems, while also being able to trap bad states and deadlocks in hard-to-reach areas of such models.

Keywords: Fuzzing techniques · Model Checking · Parameterized Systems

1 Introduction

Cubicle [5, 3] is a model checker for verifying safety properties of array-based systems. This is a syntactically restricted class of parametrized transition systems with states represented as arrays indexed by an arbitrary number of processes (or nodes) [6]. Distributed protocols, cache coherence, and mutual exclusion algorithms are typical examples of such systems.

Cubicle is based on the Model Checking Modulo Theory (MCMT) framework [7] where states and transitions are both represented as formulas in a particular fragment of first-order logic. To verify safety, Cubicle checks that unsafe states are not reachable using a symbolic backward reachability analysis: starting from a user-defined formula describing unsafe states, it iteratively computes

its pre-image closure (understood as unreachable states), making use of an SMT back-end for termination and safety tests.

In order to speed up safety proofs, Cubicle supports invariant synthesis [4]. For that, it first computes a set $\mathcal{M}$ of reachable states using a *forward exploration* for a finite instance of the system (with a *fixed number* of processes). The current strategies implemented in Cubicle for this forward search are BFS and DFS (users can choose which strategy to use). Then, Cubicle performs a backward reachability analysis of the *parameterized* system. At each loop iteration, Cubicle computes an over-approximation of pre-images and checks that they represent states that are not in $\mathcal{M}$. All these approximations, which can be seen as *candidate invariants*, are model checked together with the original safety property. Sometimes approximations can be too coarse, leading to false positives known as spurious traces. When these occur, Cubicle is forced to backtrack in order to ensure completeness.

The strength of this method lies in the fact that finite instances are generally good oracles for guiding the choice of approximations, as they can be seen as concentrated knowledge of the system. However, the method only works if the set $\mathcal{M}$ is sufficiently large and contains crucial system states. If this is not the case, Cubicle will backtrack very often during its backward analysis, which will likely prevent it from completing its proof.

Unfortunately, the space of states $\mathcal{M}$ to be visited for a finite instance can grow exponentially, even for a small number of processes. This is the case, for example, for hierarchical systems such as cache coherence algorithms, where it is necessary to explore execution traces deep enough to visit significant states. For such systems, Cubicle's current exploration strategies are either unable to go deep enough into the system (BFS), or unable to explore subtle interleavings of component executions (DFS). In both cases, Cubicle is forced to backtrack often during its backward analysis.

In this paper, we describe an algorithm for a new forward exploration strategy for Cubicle inspired by fuzzing techniques [11, 9, 8]. This strategy not only makes it possible to explore very deep traces, but also to discover extremely rare events in a system, such as synchronization points resulting from highly improbable interleavings. The relevance of the states visited by this approach is such that it enables Cubicle to deduce invariants for systems that previously ranged from difficult to impossible to analyze. Furthermore, not only does this new exploration technique provide Cubicle with a bigger variety of states for inferring invariants, it also quickly identifies unsafe models. As a bonus, it adds testing capabilities to Cubicle, such as the ability to detect deadlocks.

To summarize, we make the following contributions:

1. We define the Cubicle Fuzzy Loop (CFL), a new (forward) exploration algorithm for Cubicle based on fuzzing techniques, for which we present and discuss different heuristics.
2. We have implemented CFL in a new prototype version of Cubicle. We are experimentally evaluating the benefits of CFL on representative examples of highly concurrent and hierarchical systems.

3. Finally, we demonstrate experimentally that CFL can be easily extended to detect deadlocks, which is not possible with the current version of Cubicle.

The rest of the paper is organized as follows: In Section 2, we recall the backward reachability algorithm of Cubicle and its (candidate) invariant inference mechanism. In Section 3, we illustrate how CFL works on a simple example that is representative of systems that are difficult for Cubicle to analyze. We formalize CFL in Section 4. We show and discuss experimental results in Section 5. We conclude and present related works in Section 6.

2 Backgound on Cubicle

Cubicle is based on MCMT, a declarative framework for parameterized systems in which (sets of) states, transitions and properties are expressed in a particular fragment of first order logic with enumerative data types. Systems expressible in this framework are called array-based transition systems, because their states can be seen as a set of unbounded arrays (denoted by capital letters $\mathsf{X}, \mathsf{Y}, \ldots$) whose indexes range over elements of a parameterized domain, called proc, of process identifiers (denoted by $i, j, ...$). Given an array variable X and a process variable i, we write $\mathsf{X}[i]$ for an array access of X at index i. Systems may also contain variables but, from a theoretical point of view, a variable is seen as an array with the same value in all its cells. Arrays may contain integers or real numbers, booleans (or constructors from an enumerative user-defined datatype), or process identifiers.

A parameterized array-based system $\mathcal{S}$ is defined by a triplet $(\mathcal{X}, I, \tau)$ where $\mathcal{X}$ is a set of array symbols, I is a formula describing the initial states of the system and τ is a set of (possibly quantified) formulas, called *transitions*, relating states of $\mathcal{S}$. The formula I is a universal conjunction of literals of the form $\forall \boldsymbol{i}. \bigwedge_n \ell_n$ which characterizes the values for some array entries. Each literal ℓ_n is a comparison ($=, \neq, <, \leq$) between two terms. A term can be a constant (integer, boolean, real, constructor), a process variable (i), an array access $\mathsf{X}[i]$. A transition $t \in \tau$ is represented by a formula parameterized by the set of variables before and after the transition ($\mathcal{X}$ and $\mathcal{X}'$) and prefixed by the existentially quantified process variables involved in the transition:

$$t(\mathcal{X}, \mathcal{X}') = \exists \boldsymbol{i}.\, \Delta(\boldsymbol{i}) \wedge \gamma(\boldsymbol{i}, \mathcal{X}) \\ \wedge \bigwedge_{\mathsf{X}' \in \mathcal{X}'} \forall k.\, \bigwedge_n \left(C_n(\boldsymbol{i}, k, \mathcal{X}) \Rightarrow \mathsf{X}'[k] = v_n(\boldsymbol{i}, k, \mathcal{X})\right)$$

where $\Delta(\boldsymbol{i})$ is the conjunction of all disequations between the variables in $\boldsymbol{i}$, the formula $\gamma(\boldsymbol{i}, \mathcal{X})$ is a conjunction of literals that represents the transition's guard, *i.e.* the conditions that must be met for the transition to be triggered and the conjunction $\bigwedge_n (C_n(\boldsymbol{i}, k, \mathcal{X}) \Rightarrow \mathsf{X}'[k] = v_n(\boldsymbol{i}, k, \mathcal{X}))$ represents the updated value of each array X defined by a case-split expression, where each conjunction of literals $C_n(\boldsymbol{i}, k, \mathcal{X})$ and term $v_n(\boldsymbol{i}, k, \mathcal{X})$ may depend on $\boldsymbol{i}, k$ and $\mathcal{X}$.

In Fig. 1, we give an example of an array-based system implementing a simple, slightly modified, Dekker mutual exclusion algorithm. The system keeps track of the status $\mathtt{S}[i]$ of a process i. A process can have one of three statuses:

- `Idle`: the process is not doing anything in particular
- `Want`: the process has requested access to the critical section
- `Crit`: the process has been granted access to the critical section

As denoted by the formula Init in Fig. 1, the status of every process i is Idle in the initial state of the system. There is also a variable Turn, keeping track of who among those who've requested access can enter the critical section (the content of Turn is not specified in the Init formula). The three transitions Req, Enter and Exit describe the behavior of any process i. For example, transition `Enter` should be read as: if there exists a process i such that $\mathsf{S}[i] = \texttt{Want}$ and $\mathsf{Turn} = i$, then the new value of the array S, called S', is $\mathsf{S}[i \leftarrow \texttt{Crit}]$ which succinctly denotes an array equal to S, except for cell i, which is now equal to `Crit`.

type t = `Idle` | `Want` | `Crit`
globals:
Turn : proc
S : (proc, t) array

Init:
$\forall i.$ $\mathsf{S}[i] = \texttt{Idle}$

Unsafe:
$\exists i,j.$ $i \neq j \wedge \mathsf{S}[i] = \texttt{Crit} \wedge \mathsf{S}[j] = \texttt{Crit}$

Req: $\exists i.$ $\mathsf{S}[i] = \texttt{Idle} \wedge \mathsf{Turn} = i \wedge \mathsf{S}' = \mathsf{S}[i \leftarrow \texttt{Want}] \wedge \mathsf{Turn} = \mathsf{Turn}'$

Enter: $\exists i.$ $\mathsf{S}[i] = \texttt{Want} \wedge \mathsf{Turn} = i \wedge \mathsf{S}' = \mathsf{S}[i \leftarrow \texttt{Crit}] \wedge \mathsf{Turn} = \mathsf{Turn}'$

Exit: $\exists i,j.$ $\mathsf{S}[i] = \texttt{Crit} \wedge \mathsf{S}' = \mathsf{S}[i \leftarrow \texttt{Idle}] \wedge \mathsf{Turn}' = j$

Fig. 1. Modified Dekker mutual exclusion algorithm

Safety properties to be verified on array-based systems are expressed in their negated form as formulas that represent unsafe states. Each unsafe formula $\varphi(\mathcal{X})$ must be a *cube*, i.e., have the form $\exists \boldsymbol{k}.(\Delta(\boldsymbol{k}) \wedge \bigwedge_m \ell_m(\boldsymbol{k}, \mathcal{X}))$, where each literal $\ell_m(\boldsymbol{k}, \mathcal{X})$ may depend on $\boldsymbol{k}$ and array symbols in $\mathcal{X}$. For example, the Unsafe formula in Fig. 1 describes the bad states of the Dekker algorithm, which correspond to states where two distinct processes have been granted access to the critical section simultaneously.

For a state formula φ and a transition $t \in \tau$, let $pre_t(\varphi)$ be the formula describing the set of states from which a φ-state can be reached in one t-step. The pre-image of a formula $\varphi(\mathcal{X})$ by a transition t is given by:

$$pre_t(\varphi)(\mathcal{X}) = \exists \mathcal{X}'.\, t(\mathcal{X}, \mathcal{X}') \wedge \varphi(\mathcal{X}')$$

The pre-image *closure* of φ w.r.t a set of transitions τ, denoted by $\textsc{Pre}^*_\tau(\varphi)$, is defined as follows:

$$\begin{cases} \textsc{Pre}^0_\tau(\varphi) \triangleq \varphi \\ \textsc{Pre}^n_\tau(\varphi) \triangleq \bigcup\{pre_t(\psi) \mid \psi \in \textsc{Pre}^{n-1}_\tau(\varphi), t \in \tau\} \\ \textsc{Pre}^*_\tau(\varphi) \triangleq \bigcup_{k \in \mathbb{N}} \textsc{Pre}^k_\tau(\varphi) \end{cases}$$

and the pre-image of a set of formulas V is defined by $\text{PRE}^*_\tau(V) = \bigcup_{\varphi \in V} \text{PRE}^*_\tau(\varphi)$. We also write $\text{PRE}_\tau(\varphi)$ for $\text{PRE}^1_\tau(\varphi)$.

Given an array-based parameterized system $\mathcal{S} = (\mathcal{X}, I, \tau)$ and a set of unsafe states represented by a cube U, we say that U is *reachable* if and only if $\text{PRE}^*_\tau(V) \wedge I$ satisfiable. In order to decide if U is reachable or not, Cubicle implements the symbolic backward reachability loop $\mathsf{Bwd}(\mathcal{S}, U, d_{\max}, k)$ given in Algorithm 1. This function takes as input a parameterized system $\mathcal{S}$, a cube U, and two integers $d_{\max}$ and k. It starts by initializing a variable $\mathcal{M}$ with the set $\texttt{FWD}(d_{\max}, k)$ of reachable states constructed by a forward exploration of the reachability graph for k processes starting in a state defined by the formula $I(\#1) \wedge \cdots \wedge I(\#k)$ and limited to depth $d_{\max}$. `FWD` is not fixed and can be any user-chosen forward exploration strategy (BFS, DFS, etc).

Algorithm 1: Cubicle backward reachability loop

```
1  function Bwd(S, U, d_max, k) : begin
2      M := FWD(d_max,k);
3      V := ∅;
4      push(Q, U);
5      while not_empty(Q) do
6          φ := pop(Q);
7          if φ ∧ I satisfiable then
8              return unsafe
9          else if φ ⊭ V then
10             V := V ∪ {φ};
11             ψ := Approx(φ);
12             if M ⊭ ψ then
13                 push(Q, Pre_τ(ψ))
14             else
15                 push(Q, Pre_τ(φ))
16     return safe
```

Then, $\mathsf{Bwd}(\mathcal{S}, U, d_{\max}, k)$ computes the pre-image closure of U by maintaining two collections of states:

- Q contains the (unsafe) states to visit (it is initialized with U)
- V is filled with the visited states (initially empty)

Each iteration of the loop performs the following operations:

1. (*pop*) retrieve and remove a formula φ from Q
2. (*safety test*) check the satisfiability of $\varphi \wedge I$, *i.e.* determine if the states described by φ intersect with the initial states I. If so, the system is declared as *unsafe*

3. (*fixpoint test*) check if $\varphi \models V$ is valid, *i.e.* determine if the states described by φ have already been visited. If so, discard φ and go back to 1
4. (*over-approximate*) call function `Approx` to find an over-approximation of φ.
5. (*oracle test* and *pre-image*) if ψ represents states that are not in $\mathcal{M}$ (check the validity of $\mathcal{M} \not\models \psi$), then compute the pre-image $\mathsf{Pre}_\tau(\psi)$ of ψ and add these new (set of) states to Q. Otherwise, compute the pre-image $\mathsf{Pre}_\tau(\varphi)$ of φ and add the result to Q.

If Q is empty at step 1, then all of the state space has been explored and the system is declared *safe*. Note that the (non-trivial) fixpoint and safety tests are discharged to an embedded SMT solver. Notice that the *correctness* of Bwd does not depend on the content of $\mathcal{M}$, which thus acts as an oracle and only impacts the completeness of the algorithm.

3 Motivation

Cubicle's current forward exploration strategies are extremely efficient, but have their limitations. In this section we show how and where Cubicle struggles.

If we consider real-life concurrent systems and how they are built, there are three prevailing features: (i) pipeline parallelism, (ii) synchronization barriers, and (iii) nondeterminism. Pipeline parallelism breaks up a task into a sequence of sub-tasks, where each one can be treated concurrently by the system. This is done to improve performance by leveraging parallel processing. It complicates system models, because it not only adds depth, since each sub-task becomes an independent transition, it also introduces more interleavings to check. Synchronization barriers are necessary to coordinate the multiple processes in a concurrent system. For example processes may be required to be in a certain configuration before gaining access to specific parts of the system. These conditions can be very precise, which can lead to them appearing rarely. Last but not least, nondeterminism is inherent to concurrent systems- processes can behave independently or run tasks in parallel, and the order in which they do this can differ from execution to execution, which again adds multiple branchings to a model.

We condense these features into a specific pattern, shown in Fig. 2. There we can see an initial node (at the top) with multiple arrows leading from it. This is to simulate branching and nondeterminism, since a process at that stage would be able to choose any of the arrows. After branching, we insert the pipeline - multiple transitions to represent a task. This adds depth to our models. Note that at any point, when a process gets finished with a sub-task, it can decide to either continue forward to the next task, or go back. All of this culminates with a synchronization barrier that demands processes behave a certain way to be activated. It is important to note that while we constructed our pattern in this order, in real life the elements can appear wherever and however often they want. This pattern can also repeat itself, leading to hierarchical systems.

The problem is that this specific pattern and its repetition, so prevalent in concurrent systems, is exactly at the root of Cubicle's limitations. We converted

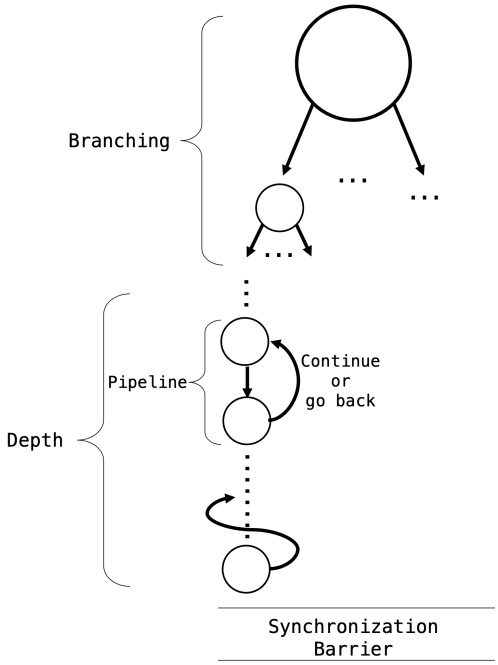

Fig. 2. Concurrent systems pattern

our pattern to Cubicle types and transitions, shown in Fig. 3. The `branch` transitions are to give a process initial choices. The transitions `pipeline` and `task` simulate breaking up one task into multiple sub-tasks. Note that these transitions can be repeated many times to complicate the system. We give an example synchronization barrier transition `sync`. This transitions's guard can easily range from simple to more complex. When faced with this pattern, both of Cubicle's forward strategies face difficulties. BFS will be forced to run through every possible branching before being able to go down a level. The more branchings there are, combined with an elevated number of processes, the longer BFS has to spend checking every one. And as we stated, this pattern can repeat itself, so the interesting part of the system might be below the synchronization barrier, but BFS will visit countless states before it even gets close to it. DFS handles this specific problem better than BFS, as it privileges depth. But complicated interleavings and algorithms that do not loop slow it down and lower its efficiency.

For example, we take the previous pattern and create a model for three processes. We give a process four initial `branch` transitions (*i.e.* $k = 4$ in Fig. 3), as well as four tasks decomposed into three sub-tasks each (*i.e.* $m = 3$ and $n = 4$ in Fig. 3) and set a synchronization barrier that forces each of the three process to be doing different tasks in order to be activated. We let BFS and DFS each explore 1 000 000 states to see how often they visit the synchronization barrier. This is important because activating the barrier means having access to the

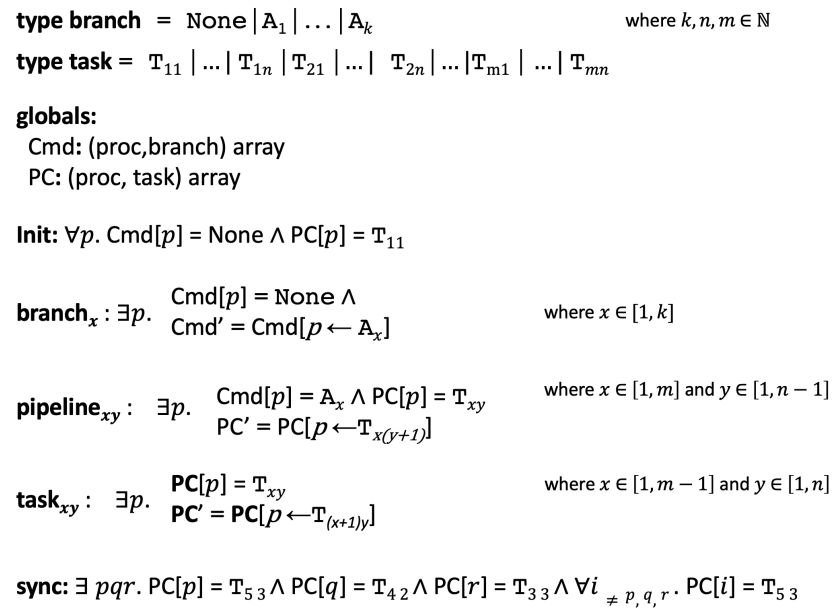

Fig. 3. Pattern as Cubicle transitions

potentially interesting transitions behind it. For 1 000 000 visited states, both BFS and DFS visited the `sync` transition two times.

We turn to fuzzing techniques to mitigate this problem. CFL's goal is to tackle this pattern by basically abandoning exhaustivity and skipping around the system. CFL abandons exhaustivity because it does not try to methodically explore every single path in the system - it tries to diversify the state space as much as possible. The reason it skips around the system is that anytime a state is visited by CFL, this state becomes an eligible initial state from which CFL can explore. This means that CFL has a higher chance of directly accessing crucial states and exploring from them. If we let CFL explore 1 000 000 states for the above example, it visits the `sync` transition approximately 150 times.

4 Fuzzing Cubicle

In this section we discuss and formalize CFL, detailing how we draw from fuzzing to create a new exploration strategy for Cubicle.

Fuzzing is essentially rapidly generating inputs for a program to see how it reacts. If an input leads to new code coverage, that input is retained and later mutated to generate new inputs that hopefully lead to more new code coverage. We retain two key notions – *new inputs* and *mutation* – both of which we want to incorporate into Cubicle. This is not straightforward, because Cubicle directly contradicts both these notions.

Cubicle's models have fixed initial states, meaning that any system exploration starts from there. We cannot randomly generate these states, since we cannot guarantee reachability. We also cannot take reachable states and mutate them for the same reasons. To fix the input problem, CFL takes already visited states and reuses them as the initial state. This guarantees that all initial states are reachable. It also allows us to diversify the explored state space: any visited state can become the initial state from which a system exploration is run.

However, setting the initial state isn't enough. When inputs are mutated in a fuzzer, the hope is that it will lead to new coverage and/or behavior fast. Simply setting new initial states in Cubicle does not lead to that if the exploration itself is not modified. The problem with the DFS and BFS strategies as they are now in Cubicle is that they are exhaustive and provide no feedback while they run, whereas we want something that might not provide exhaustivity, but will skip around the system trying to visit as many interesting new states as possible. This is why we have decided that since we cannot mutate states, we will *mutate the scheduler*, i.e. change exploration tactics while CFL runs. CFL has multiple exploration techniques, and each time an initial state is chosen, one of these techniques is run. Before going into detail on the techniques themselves, it is first necessary to describe how CFL treats states.

In CFL each state s is represented as a CFL node, a record containing the following fields:

- `state`: the explicit representation of s where variables (or arrays) are mapped to their values
- `count`: the number of times s has been visited
- `exit_num`: the number of *exit* transitions from s, *i.e.* the transitions with guards evaluating to true in s
- `exit_transitions`: an explicit representation of the *exit* transitions from s (represented by the name of the transitions and their arguments)
- `exits_taken`: which transitions have not yet been taken from s
- `exit_count`: how many times each *exit* transition has been taken.

CFL essentially keeps track of two key pieces of information: a map $\mathcal{V}$ of visited explicit states mapped to their corresponding nodes, and a set $\mathcal{P}$ of potential initial fuzzer nodes. Any time a new explicit state is visited its calculated fuzzer node to is added to $\mathcal{P}$ and the mapping of the explicit state to the node is added to $\mathcal{V}$.

The reason we keep track of *exit transitions* is because they decide when a node is no longer an interesting initial candidate. If every potential exit transition has been taken, then that node can no longer offer any new information and can be removed from $\mathcal{P}$. The basic algorithm for CFL is given in Algorithm 2.

Initially, $\mathcal{V}$ and $\mathcal{P}$ start off empty. CFL explores the model for a given number of processes k. It calculates all possible transitions for all processes on line 3. For example if the model only contains a transition `t(i)` and CFL is run with three processes, $\mathcal{T}$ with contain `t(#1)`, `t(#2)`, and `t(#3)`. It does the same for the unsafe formulas on line 4. The user-declared initial state is instantiated for k processes on line 5 and is then added to $\mathcal{P}$.

Algorithm 2: Basic CFL Algorithm

```
1  V := ∅ ;
2  P := ∅ ;
3  T := init_transitions(k);
4  U := all_unsafes(k);
5  Init := init_system(k);
6  P := P ∪ {Init};
7  while not_empty(P) do
8  |   n := choose_node(P);
9  |   explore := choose_strategy();
10 |   V,P := explore(n, U, T);
11 end
```

CFL then takes the form of a while loop that runs as long as there are still potential initial nodes to process in $\mathcal{P}$. During the loop, it first chooses a random node from $\mathcal{P}$, chooses a random exploration technique (described below), and applies the technique to the node. Both $\mathcal{V}$ and $\mathcal{P}$ are modified as a result of this. When choosing a random exploration technique, CFL has the choice between six techniques, detailed below.

1. *Random exploration*: CFL chooses a number of steps and applies random transitions to the starting node for that many steps.
2. *Process sequences*: CFL selects a random process, picks a number of steps, and only moves that process forward for that amount of steps (or until it can't anymore)
3. *Weighted decision*: CFL grades potential steps using the following criteria
   - this step will lead me to a never visited state
   - this step means taking a transition that has never been taken by anyone globally
   - this step means taking a transition never taken from this node

   These criteria are in order of importance - being able to visit a state that has never been visited will outweigh the rest.
4. *Maximizing randomness*: a certain percentage of the time, CFL picks steps that will give the most choices in the next step.
5. *Limited BFS*: runs a very limited depth BFS from the node
6. *Unused exit*: covers an exit that hasn't been taken yet

Each technique follows the same basic algorithm, shown in Algorithm 3. It first picks a random number s of steps (*bound* can be set by the user) to take and sets the current step *curr* to zero. The environment *env* is set to the chosen node, and all possible transitions from that node's explicit state are kept in *poss*. Then, while the current number of steps taken is less than the chosen s, each technique does the following: on line 5, it picks a transition from all possible transitions according to the current technique. So for example if the current technique is *Process sequences* and the chosen process is #1, `technique` will return a transition with #1 as an argument.

Algorithm 3: Basic exploration technique template

```
1  function explore(n, 𝒰, 𝒯) : begin
2      s := random_int(bound); curr := 0; env := n;
3      poss := env.exit_transitions;
4      while curr < s do
5          t := technique(poss);
6          clean_exits(env, t);
7          state := apply_transition(env, t);
8          check_unsafe(state, 𝒰);
9          try:
10             env := find(state, 𝒱);
11             env.count := env.count + 1;
12             poss := env.exit_transitions;
13         catch NotFound:
14             poss := all_possible_transitions(state, 𝒯);
15             env := init_node(state, poss);
16             𝒱 := add(env, state, 𝒱);
17             𝒫 := 𝒫 ∪ {env};
18         end
19     end
20 end
```

CFL cleans the aforementioned *exit transitions* in the fuzzer node on line 6. For example if t is a transition that's never been taken, the `exits_taken` field will be modified in the node to include t. Then at line 7, the transition is applied to the node and a new explicit state, *state*, is calculated. Line 8 checks *state* against the unsafe formulas. How this is treated depends on how CFL is being run. If the algorithm is running for proving safety, then encountering an unsafe state immediately makes Cubicle return `Unsafe`. If CFL is running in a standalone fashion, it only shows a warning, but does not stop. Then (lines 9-18) the algorithm checks if a mapping from *state* to a node already exists in $\mathcal{V}$. If it does, then *env* is set to the existing node, with only its `count` being modified and *poss* is set to the possible exits from that node. If a mapping doesn't exist, then *poss* is calculated, a fuzzer node is created, a mapping is added to $\mathcal{V}$ and the node is added to $\mathcal{P}$. When a node is initialized, `count` is set to 1, `exit_num` is set to how many transitions are in *poss*, `exit_transitions` is set to *poss*, `exits_taken` is empty, and `exit_count` has 0 for every possible transition.

5 Experimental Results & Discussion

CFL is implemented in Cubicle[1]. As mentioned in Section 3, there is a specific recurring pattern in strongly concurrent and hierarchical models. This pattern

[1] https://github.com/cubicle-model-checker/cubicle/tree/debugger

simply serves as a template and we build off of it to test CFL and run our benchmarks.

We compare several forward exploration strategies with our new CFL heuristic: (i) Cubicle's existing BFS and DFS strategies, both optimized for speed, (ii) a random exploration strategy, i.e. one that starts at the intial state and randomly chooses transitions, and (iii) CMurphi, an enumerative model checker [12] developed on top of Murφ, only used here to efficiently visit the state space. The results of this comparison, excluding CMurphi, can be seen in Table 1. We discuss CMurphi separately further down.

Model	Forward Time	BFS			DFS			Random			CFL		
		States	Safe	Total Time	States	Safe	Total Time	States	Safe	Total Time	States	Safe	Total Time
Dekker	10s	466K	T.O.	-	605K	Yes	12.72s	266K	Yes	11.74s	120K	Yes	10.61s
Germanish	10s	424K	T.O.	-	593K	Yes	12.91s	261K	Yes	11.94s	120K	Yes	10.78s
Germanish2	10s	315K	T.O.	-	515K	Yes	12.26s	244K	Yes	11.92s	115K	Yes	10.75s
Germanish4	10s	287K	T.O.	-	547K	Yes	14.54s	186K	T.O.	-	110K	Yes	11s
German	10s	312K	T.O.	-	547K	Yes	16.25s	207K	Yes	13.55	107K	Yes	12.23s
German_Baukus	10s	359K	T.O.	-	591L	Yes	14.82s	207K	Yes	12.93s	105K	Yes	12s
German_CTC	50s	1 429K	T.O.	-	2 010K	Yes	62.81s	505K	T.O.	-	265K	Yes	55.17s
German_pfs	10s	416K	T.O.	-	431K	Yes	17.37s	174K	Yes	12.69s	100K	Yes	13.11s
Szymanski_at	10s	372K	T.O.	-	534K	T.O.	-	155K	Yes	11.92s	105K	Yes	11.60s
Szymanski_na	10s	270K	T.O.	-	483K	T.O.	-	270K	T.O.	-	100K	Yes	12.50s
Bakery_lamport	40s	1 565K	T.O.	-	2038K	T.O.	-	650K	T.O.	-	230K	Yes	42.59s
Flash_no_data	40s	862K	T.O.	-	1 048K	T.O.	-	273K	T.O.	-	140K	Yes	43.32s

Table 1. Comparing CFL with different forward strategies.

Each strategy is run for three processes and has the same amount of time allocated for its forward exploration, noted in the Forward Time column. We then compare how many states were visited (States column) and whether Cubicle was able to prove safety before hitting the timeout criteria (Safe column). The total time (forward + proof) is noted in the Total Time column for each strategy. Each example was timed out after 5 minutes. This was chosen due to the time taken using CFL, as well as the number of proof nodes generated by Cubicle within those 5 minutes, compared in Table 2. The values underlined and in bold are where Cubicle was successful in proving safety. We can see that the number of nodes for the timed out examples is much higher than is necessary for Cubicle in the cases where it quickly proves safety.

Another problem is that, when it comes to Cubicle, models following patterns like the one described above are a double-edged sword. When they are safe, a proof will take a long time, and when they are unsafe, a counter-example might also take a long time. Both of these things are impacted by the number of states visited during the forward exploration. More visited states does not necessarily imply a faster proof, since Cubicle will have to compare its invariant candidates to every state. The key is visiting fewer, but more important, states. Cubicle is designed to prove safety, and while it *will* give a counter-example should the system be unsafe, this can take an arbitrarily long time in huge systems. The

Model	BFS	DFS	Random	CFL
Dekker	6904	**4**	**4**	**4**
Germanish	889	**4**	**4**	**4**
Germanish2	1770	**4**	**4**	**4**
Germanish4	2415	**20**	3255	**20**
German	2862	**41**	**41**	**41**
German_Baukus	2170	**41**	**41**	**41**
German_CTC	1500	**61**	1231	**60**
German_pfs	1121	**44**	**44**	**44**
Szymanski_at	2861	174	**33**	**33**
Szymanski_na	2061	210	510	**43**
Bakery_lamport	779	2189	230	**16**
Flash_no_data	1329	61	1227	**37**

Table 2. Number of generated proof nodes for each strategy

forward and backward algorithm face the same problem in essence- huge safe models take too much time to explore forward, and huge unsafe models take too much time to trace backward. Running a time-and-calculation-heavy proof only to be hit with an "`Unsafe`" for trivial reasons is something we want to avoid. This problem is in the same family as trying to prove safety when the model deadlocks. When Cubicle says that a model is safe, it is safe - there is no way to get from the initial state to the unsafe state. However, the reason for that could be a correctly written model, or a model that deadlocks- it is natural that an unsafe state is unreachable if the model is incapable of taking any steps. The inclusion of CFL in Cubicle allows us to tackle both of these problems. We buried unsafe states deep within our test models and launched CFL against Cubicle's normal backward algorithm, without any additional forward strategies to accelerate invariant finding. The results can be seen in Table 3. Once again timeout was set to five minutes. Deadlocks were a bit harder to compare - while it was fairly easy to deadlock our models, it wasn't simple to pinpoint the specific state that could be classified as a deadlock. We provide deadlock detection results for CFL in Table 4 without comparing them to Cubicle.

The reason CMurphi is excluded from Table 1 is due to the fact that we were unable to find an option that would force CMurphi to run for the allocated time. For each of our models, CMurphi raised the following error: "Internal Error: Too many active states." For the sake of fairness, we rerun CFL, manually setting the limit for each model to how many states were visited by CMurphi. The results for this are seen in Table 5.

This leads us to the discussion part of this section, namely concerning CFL's stability. As you can see in Table 5, the results for CFL all have the form `X/Y`. This is due to CFL's innate randomness. Two executions will not necessarily have the same results, especially if the allocated time/number of states to visit is low and the model is large. For example, in Table 5, Dekker was run 10 times, and all 10 times CFL managed to visit enough states to help Cubicle quickly prove safety. However, on a model like Germanish4, which is longer and more

Model	Backward	CFL
Dekker	T.O.	0.3s
Germanish	T.O.	0.7s
Germanish2	T.O.	0.2s
Germanish4	T.O.	0.7s
German	T.O.	0.4s
German_Baukus	T.O.	0.4s
German_CTC	T.O.	0.5s
German_pfs	T.O.	0.3s
Szymanski_at	T.O.	2s
Szymanski_na	T.O.	2s
Bakery_lamport	T.O.	1.5s
Flash_no_data	T.O.	3s

Table 3. Unsafe: backward vs. CFL

Model	CFL
Dekker	0.1ms
Germanish	0.5s
Germanish2	0.2s
Germanish4	0.5s
German	0.4s
German_Baukus	0.4s
German_CTC	0.4s
German_pfs	1s
Szymanski_at	2s
Szymanski_na	0.6s
Bakery_lamport	2s
Flash_no_data	4s

Table 4. Deadlock detection

complex, running CFL 10 times only led to seven quick successes. This is due to CFL containing a fair amount of randomness in how it chooses execution strategies.

Model	CMurphi		CFL	
	States	Safe	States	Safe
Dekker	48K	T.O.	48K	10/10
Germanish	48K	T.O.	48K	10/10
Germanish2	39K	T.O.	39K	10/10
Germanish4	39K	T.O.	39K	7/10
German	33K	T.O.	33K	6/10
German_Baukus	33K	T.O.	33K	7/10
German_CTC	24K	T.O.	24K	0
German_pfs	33K	T.O.	33K	6/10
Szymanski_at	32K	T.O.	32K	3/10
Szymanski_na	26K	T.O.	26K	2/10
Bakery_lamport	32K	T.O.	32K	1/10
Flash_no_data	21K	T.O.	21K	3/10

Table 5. Comparison with CMurphi

6 Conclusion and Related Work

In this paper, we presented CFL, an algorithm for a new forward exploration strategy based on fuzzing for Cubicle. CFL not only serves as an oracle for Cubicle's invariant generation algorithm, but also adds new functionalities. We show that this strategy is effective and capable of tackling a class of models that Cubicle struggles with. We describe how CFL draws from fuzzing, but is adapted to Cubicle's semantics. We show how it uses multiple exploration techniques to cover the state space as diversely as possible, leading to the discovery of crucial states needed to terminate proofs. CFL also introduces quick debugging and deadlock detection to Cubicle, quickly capturing both unsafe and deadlocking states in complicated models.

There are two immediate lines of future work. The one we are currently working on is including parameterization. The goal is for CFL to be able to estimate how many processes it needs to efficiently explore a system. The other is CFL's stability. As mentioned earlier, CFL is nondeterministic by nature, and chooses its exploration techniques randomly. Fine-tuning how these choices are made could increase CFL's performance. We also think it is important to extend CFL and add more techniques, for example allowing processes to die randomly throughout an exploration. We would also like to incorporate liveness testing into CFL, since, like with deadlocks, this would add a new functionality to Cubicle.

Our work is inspired by fuzzing. Fuzzing is a simple technique designed to quickly explore a program's execution paths. The idea of mutating and generating inputs in our case was specifically inspired by AFL [14], a state-of-the-art fuzzer. Combining model checking with fuzzing is not new. For example, the authors in [13] use it for test case generation. In [10], it serves as the inspiration to test Linear-time Temporal Logic (LTL) properties for C++ programs. Bounded model checking (BMC) has been combined with fuzzing in multiple instances. For example in [2], BMC is used to generate paths that the fuzzer would not have found on its own. In [1], the authors combine BMC and Gray-Box Fuzzing to find vulnerabilities in concurrent programs. To our knowledge, no previous works combine fuzzing with parameterized model checking. Our end-goal also diverges, the above examples all dealing with actual code, whereas we want to focus on the model. We consider this to be a new line of research, perfectly suited for Cubicle, since Cubicle's invariant generation needs a forward exploration strategy that is not exhaustive (contrary to model checking) but is capable of exploring the state space efficiently.

References

1. Aljaafari, F.K., Menezes, R., Manino, E., Shmarov, F., Mustafa, M.A., Cordeiro, L.C.: Combining bmc and fuzzing techniques for finding software vulnerabilities in concurrent programs. IEEE Access **10**, 121365–121384 (2022)
2. Alshmrany, K.M., Aldughaim, M., Bhayat, A., Cordeiro, L.C.: Fusebmc v4: Smart seed generation for hybrid fuzzing: (competition contribution). In: Fundamental Approaches to Software Engineering: 25th International Conference, FASE 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings. pp. 336–340. Springer International Publishing Cham (2022)
3. Conchon, S., Goel, A., Krstić, S., Mebsout, A., Zaïdi, F.: Cubicle: A parallel smt-based model checker for parameterized systems: Tool paper. In: CAV. pp. 718–724. CAV'12, Springer-Verlag, Berlin, Heidelberg (2012)
4. Conchon, S., Goel, A., Krstić, S., Mebsout, A., Zaïdi, F.: Invariants for finite instances and beyond. In: 2013 Formal Methods in Computer-Aided Design. pp. 61–68. IEEE (2013)
5. Conchon, S., Mebsout, A., Zaïdi, F.: Vérification de systèmes paramétrés avec Cubicle. In: JFLA. Aussois, France (Feb 2013), http://hal.inria.fr/hal-00778832
6. Ghilardi, S., Nicolini, E., Ranise, S., Zucchelli, D.: Towards SMT model checking of array-based systems. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) Automated Reasoning, Lecture Notes in Computer Science, vol. 5195, pp. 67–82. Springer Berlin Heidelberg (2008)
7. Ghilardi, S., Ranise, S.: MCMT: A model checker modulo theories. In: IJCAR. pp. 22–29 (2010)
8. Godefroid, P.: Fuzzing: Hack, art, and science. Communications of the ACM **63**(2), 70–76 (2020)
9. Manès, V.J., Han, H., Han, C., Cha, S.K., Egele, M., Schwartz, E.J., Woo, M.: The art, science, and engineering of fuzzing: A survey. IEEE Transactions on Software Engineering **47**(11), 2312–2331 (2019)

10. Meng, R., Dong, Z., Li, J., Beschastnikh, I., Roychoudhury, A.: Linear-time temporal logic guided greybox fuzzing. In: Proceedings of the 44th International Conference on Software Engineering. pp. 1343–1355 (2022)
11. Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of unix utilities. Communications of the ACM **33**(12), 32–44 (1990)
12. Penna, G.D., Intrigila, B., Melatti, I., Tronci, E., Zilli, M.V.: Exploiting transition locality in automatic verification of finite-state concurrent systems. STTT **6**(4), 320–341 (2004)
13. Yang, Y.: Improve model testing by integrating bounded model checking and coverage guided fuzzing. Electronics **12**(7), 1573 (2023)
14. Zalewski, M.: American fuzzy lop-whitepaper (2016)