Abstract—Numerous devices, from network switches and servers to industrial control systems, can be unreliable if they are not configured properly. Even if a device’s implementation has been proven correct, it must still be configured to meet the specific functional and security requirements of its stakeholders.

However, manual configuration remains labor intensive and error-prone even for experts. Automated configuration synthesis presents a promising way forward. Unfortunately, as we show, existing counterexample-guided algorithms can perform poorly if the system model allows configuration changes during execution. Yet disallowing such changes can hide significant problems, such as privilege escalation.

We present a new synthesis algorithm that exploits structure inherent in state-machine models where the system configuration changes. We implement it using the Kodkod relational model finder, and show that it favorably solves a number of configuration-synthesis tasks.

I. INTRODUCTION

Numerous devices, from network switches to industrial control systems, can be unreliable and insecure if they are not configured properly. Multiple reports [1]–[3] from cloud-service providers show problems arising from misconfiguration and related issues (such as unsound configuration generation). Formal synthesis presents a promising way forward.

Most traditional synthesis projects focus on producing programs. However, even if a device’s implementation is “correct”, it will not function correctly unless it is configured to meet the user’s goals. Thus, we focus on the less-attended problem of configuration synthesis: given a fixed program, produce a configuration that will ensure those goals.

Prior work (Sec. VII) in configuration synthesis assumes that the system configuration never changes (which certainly aids synthesis performance). In practice, this misses many real-world systems that are reconfigured by external controllers (as in software-defined networking [4]), human-executed standard operating procedures, and more. All these dynamically change the configuration. A poor initial configuration can, after mutation, become an undesirable configuration, resulting in problems like privilege escalation.

Thus, our synthesis goal is to find initial configurations that do not contain latent flaws. Our approach consumes a transition system and temporal safety goals, and generates a configuration that meets those goals even if the system mutates the configuration. Our problem domain is therefore distinct from most program synthesis (e.g., SyGuS [5]) and reactive synthesis [6], as we discuss in Sec. VII.

In this short paper, we identify the mutable-configuration synthesis problem, note issues with naively applying popular counterexample-based algorithms, and describe an approach that yields promising improvement.

II. SYNTHESIS EXAMPLE

We begin with an example domain: Role-Based Access Control [7] (RBAC). In RBAC, roles mediate user access to permissions. For instance, if user Alice has the role Accountant and the Accountant role has the ReadStatements permission, Alice can read bank statement files. An RBAC policy can be represented as a pair of relations: 

\[ \text{ra} : \text{User} \times \text{Role} \] and \[ \text{pa} : \text{Role} \times \text{Permission} \]

Administrative RBAC [8], [9] (ARBAC) formalizes online modifications to an RBAC policy. There are numerous variants of ARBAC; we adopt a simplified version for brevity (eliding, e.g., role revocation). Concretely, we add to RBAC:

\[ \text{canAssign} : \text{Role} \times \text{Role} \] says which roles users with a given administrative role are empowered to assign.

\[ \text{require} : \text{Role} \times \text{Role} \times \text{Role} \] positively filters role-assignment.

\[ \text{forbid} : \text{Role} \times \text{Role} \times \text{Role} \] negatively filters role-assignment.

For someone to use role \( a \) to grant role \( r \) to user \( u \), it must hold that

1. \( (a, r) \in \text{canAssign} \);
2. \( \forall r' \) such that \( (a, r, r') \in \text{require} \), \( (u, r') \in \text{ra} \); and
3. \( \forall r' \) such that \( (a, r, r') \in \text{forbid} \), \( (u, r') \notin \text{ra} \).

In our synthesis setting, the set of entities is bounded (although not necessarily fixed) a priori. For this example, we specify that there exist 3 potential users Alice, Bob, and Charlie along with the concrete roles Manager, Auditor, and Accountant. Now suppose that an organization has three goals:

1. The Accountant and Auditor roles are always populated by some user.
2. The Accountant and Auditor roles are always disjoint.
3. The Accountant and Auditor roles (at least) can be assigned in the initial state.

The first two goals correspond to safety properties in linear temporal logic, where states correspond to configurations. The third goal precludes trivial examples that disallow any ARBAC actions. We wish to synthesize initial values to the ARBAC relations (i.e., an initial configuration state) such that these properties hold for all potential system traces.
Mutability and Synthesis: User-role assignment is mutable: it changes over time as a result of ARBAC, and the system model used for synthesis must account for this. Moreover, in a realistic system there are other sources of mutability—both automated and human. One component’s configuration (e.g., a filesystem) may guard and enable reconfiguration of other components, and (unlike in this example) the dependencies are not necessarily acyclic. System administrators may also dynamically reconfigure live systems, limited only by standard operating procedure.

Synthesis in this setting requires an engine that is able to account for all possible system traces (i.e., higher-order universal quantification) without relying on any part of the trace to remain constant. Fortunately, as we will see, this problem contains rich structure that can be exploited to aid synthesis. Our prototype implementation generates the configuration shown in Fig. 1 in 2 seconds.

III. PRELIMINARIES

Space limitations preclude a full formalization, but we briefly establish some foundations for what follows.

Synthesis input consists of a symbolic state machine $M$ (the system model) with transition relation $\delta$ over a set of relations $R$ and a set of $R$-properties $\Theta$ in safety Linear Temporal Logic (LTL). Note that safety properties can state more than just global assertions, and allow, e.g., the weak-until operator.

As $M$ describes how configuration change can occur in the overall system, its states represent concrete system configurations. For example, the state machine for ARBAC is defined over relations $ra$, $pa$, $canAssign$, $require$, and $forbid$ and encodes the transition system for role-assignment described in Sec. II. The synthesis properties $\Theta$ are LTL formalizations of the three goals in Sec. II.

We denote the set of execution traces for $M$ as $\text{Traces}(M)$, and the set of length-$n$ traces as $\text{Traces}_n(M)$. If all traces starting at $s$ satisfy $\Theta$ then we say that $s$ satisfies $\Theta$. If all traces of length $n$ starting at $s$ (extended to infinite-length via stuttering) satisfy $\Theta$, then we say that $s$ is $n$-feasible for $\Theta$.

The goal of synthesis is to provide a configuration (i.e., a start state of $M$) that satisfies $\Theta$. For the Sec. II example, Fig. 1 gives a configuration that satisfies all 3 goals shown. For simplicity, we consider the bounded-trace version of synthesis: given a machine $M$, a finite set of safety requirements $\Theta$ and a trace-length $n$, we seek an $n$-feasible configuration of the machine for $\Theta$. Logically, this means satisfying $\exists c \forall s'. (s_0 = c \land s \in \text{Traces}_n(M)) \implies \Theta(s')$, where $c$ ranges over configurations and $s'$ over traces, and $s_0$ denotes the first state in $s$.

IV. ALGORITHMICS

A. A First Attempt: Learning from counterexamples

Such problems often submit to the CounterExample-Guided Inductive Synthesis (CEGIS) approach. The key idea is that, while the $\exists$ quantifier may be expensive to eliminate, counterexamples for $\Theta$ starting from a fixed candidate $c$ are easier to produce and so a solution can be reached by successive approximation.

Adapted for our domain, CEGIS proceeds in two parts: (1) find a candidate $c$ that satisfies $\exists c \forall s' \in S. (s_0 = c \land s \in \text{Traces}_n(M)) \implies \Theta(s')$ for small (initially empty) $S$; then (2) attempt to satisfy $\exists s'. s_0 = c \land s \in \text{Traces}_n(M) \land \neg \Theta(s')$. Should phase (2) yield a counterexample trace, it is added to $S$ and phase (1) repeats. Otherwise, $c$ is correct.

Adding a fixed trace $s'$ to $S$ involves learning the instantiation of the quantifier-free subformula with the trace. Since $\neg \Theta(s')$ always holds, this means excluding the counterexample from the system’s behavior. But $\text{Traces}_n(M)$ is fixed, so the new constraint is just $s_0 \neq c$. Thus, a naive iteration of CEGIS will only learn that a different configuration is necessary. Given the large search space of potential configurations (the small example in Sec. II admits $2^{81}$ since there are 81 potential tuples), we must do better.

B. A Way Forward: Extracting Blame

To improve on the above algorithm, we exploit two key ideas. First, if a trace violates $\Theta$, there will be a finite set of relational facts (possibly spanning multiple states in the trace) that suffice to violate $\Theta$. Second, these facts describe a set of traces that can be transformed under the transitions of $M$ to yield a set of shorter traces that can be extended to violate $\Theta$. The revised algorithm, Alg. II adds corresponding new phases 3 and 4 to CEGIS. We address each phase individually.

- **Phase 1** (Line 3–Line 5): Find a candidate starting state $s_0$ via a satisfiability query on the initial-state formula $\alpha_0$ of the system $M$ plus any constraints learned so far. If this query is unsatisfiable, all possible configurations have been excluded. Otherwise, obtain candidate configuration $c$ from the result.

- **Phase 2** (Line 6–Line 7): Seek a trace starting from $c$ that violates the requirements $\Theta$. The only subtlety is that we must reframe the problem to return traces rather than a starting configuration. We use $M$ to denote the formula that encodes $\text{Traces}_n(M)$. We translate finite-trace LTL formulas to first-order logic in the usual way. For instance, the statement that $XGp$ ($p$ holds globally from the second state) would become the assertion that, for all state atoms after the first, $p$ holds.

If there is no counterexample, return the $n$-feasible configuration. Otherwise, another iteration is needed.
Algorithm 1: Configuration Synthesis

\begin{algorithm}
\begin{algorithmic}[1]
\State $L \leftarrow \text{empty}$;
\While {true}
\State $s_0 \leftarrow \text{solve}(\alpha_0 \land L)$;
\If {unsat$(s_0)$} \textbf{then} \Return fail(); \EndIf
\State candidate $\leftarrow s_0$;
\State $\delta \leftarrow \text{solve}(\text{candidate} \land \overline{\delta})$;
\If {unsat$(\delta)$} \textbf{then} \Return success(candidates); \EndIf
\State $w' \leftarrow \text{minimize}(\text{solve}(\text{poststate} \land \text{prestate}))$;
\State $w' \leftarrow \text{filter}(\text{solve}(\text{poststate} \land \text{prestate}))$; \Comment{bucket by state}
\State $\delta \cdot w \leftarrow \overline{\delta} \land w$; \Comment{discard final state}
\While {$|\overline{w}| > 0$}
\State $\overline{\delta} \cdot \overline{\delta} \leftarrow \overline{\delta} \cdot w$; \Comment{isolate current prestate}
\State $\overline{\delta} \cdot \overline{\delta} \cdot \overline{\delta} \leftarrow \overline{\delta} \cdot w'$; \Comment{isolate current poststate}
\State $\delta \leftarrow \text{filter}(\text{minimize}(\text{solve}(\text{prestate} \land \text{poststate}))$;
\State $\delta \cdot \delta \leftarrow \delta \cdot (\overline{w} \land \overline{w})$; \Comment{empty \delta is treated as true}
\State $\overline{\delta} \cdot w \leftarrow \overline{\delta} \cdot w'$; \Comment{discard last remaining component of \overline{w}}
\EndWhile
\State $L \leftarrow L \land \overline{\delta}$;
\EndWhile
\end{algorithmic}
\end{algorithm}

- **Phase 3** (Line 8–Line 9): Determine a proximal cause. Given a counterexample trace $\overline{s}$, we seek a smallest justification for its failure on $\Theta$. While the conjunction of all facts in $\overline{s}$ suffices, this is often much larger than needed. Consider the LTL formula $G_{p_{100}}$ ("the proposition $p_{100}$ is true globally"). There may be many variables in a counterexample trace, but the lone fact that it includes a state where $p_{100}$ is false suffices as a **proximal cause** for property failure.

Alg. 1 obtains a minimal proximal cause by asking the solver to satisfy $\overline{s} \land \Theta$, where $\overline{s}$ is a conjunction of literals that fully describe the counterexample trace. The overall query must be unsatisfiable, or else $\overline{s}$ would not be a counterexample. By using a core-extracting solver (Sec. V), our algorithm can obtain a minimal unsatisfiable core: a minimal subset of the conjuncts that are themselves unsatisfiable. Filtering the core to only literals of $\overline{s}$ and sorting its components by state yields a vector of formulas $\overline{w}$ that the next phase will, intuitively, refine into a root cause in the initial state.

- **Phase 4** (Line 10–Line 18): Determine the root cause. To do this, Alg. 1 iteratively reduces a sequence of enabling causes for $\neg \Theta$, beginning with the proximal cause.

In each inner iteration the length of the enabling cause $\overline{w}$ is decreased, eventually terminating at a cause entirely in terms of the initial state. For any initial state $s$ that satisfies this final enabling cause, some trace starting at $s$ exists that violates $\Theta$, and thus the algorithm can soundly learn the cause’s negation after filtering to the candidate. (Showing that this is sound involves proving by induction that, essentially, the causal nature of each successive $\overline{w}$ is preserved.)

If the post-filter cause $\overline{w}$ is empty, this means that there is no configuration that satisfies $\Theta$. In this case, the algorithm learns a contradiction and rightly fails in the next iteration. Learning the negation of this final formula must exclude at least one configuration, and thus Alg. 1 terminates.

Alg. 1 is no better in the worst case than 2-phase CEGIS, as it can potentially iterate once for every configuration. However, Sec. VI shows it is often of order-magnitude improvement.

V. IMPLEMENTATION

Our synthesis engine is implemented in Java and runs atop Kodkod [11]. Kodkod supports problems with rich (bounded) relational state and provides features we use such as unsat-core minimization. Below Kodkod, we use incremental Minisat for Alg. 1’s synthesis phase and proof-extracting Minisat for the other three phases. The engine accepts synthesis problems via an API, along with various options such as trace length.

VI. EVALUATION

We address the following research questions: (RQ1): Is Alg. 1 more performant than 2-phase CEGIS when synthesizing mutable configurations? (RQ2): Are the performance bottlenecks of Alg. 1? We report on the example from Sec. III (ARBAC), along with an unsatisfiable variant (Unsat) that allows a system administrator to make arbitrary changes, making the goals impossible to satisfy.

All experiments were run on a MacBook Pro (2.3 GHz i5, MacOS 10.14.4, 3 GB Java heap). Trials that did not complete in 30 minutes were terminated. Fig. 2 presents the results. The Example, Bounds, and Tr. Ln. columns indicate the example name, bounds (maximum number of users, roles, and permissions), and maximum trace length respectively.

A. RQ1: Performance vs. Baseline CEGIS

As a baseline, we modeled and ran each problem in Alloy* [13], which uses 2-phase CEGIS. The Total (s) column lists wall-clock runtime in seconds for both Alg. 1 and Alloy*, rounded to the nearest second. Alloy* failed to finish within the time limit on every Unsat example, while Alg. 1 completed the 3-role case for both trace lengths. We also report the number of iterations taken. In all cases, Alg. 1 iterates less, indicating that more information is being learned in each pass.

We ran additional experiments with an 8-hour timeout to clarify two ambiguous cases. First, we re-ran the first unsolved 5-state Unsat example for both Alg. 1 and Alloy* (bounds 4 and 3 respectively). Alg. 1 reached an unsatisfiable result in 70 minutes; Alloy* did not terminate. Second, we re-ran ARBAC(8) in Alloy*; since Alg. 1 nearly timed out at 30 minutes. Alloy* failed to terminate in 8 hours.

The Alloy* models used for comparison can be found at:

http://cs.brown.edu/research/plt/dl/seconfig2019/

B. RQ2: Performance Bottlenecks

The Phase column lists time taken, to the nearest second, per phase: synthesis, verification, proximal cause, and root cause. Sub-1-second results are shown as 1 second to avoid potentially misleading zeros. Throughout, the vast majority of runtime is spent in root-cause extraction—the step that repeatedly performs core extraction and minimization. (The gap between $s + v + p + r$ and the total time comprises overhead: setup of data structures, etc.)

The #v column reports how many disjuncts were in each iteration’s learned constraint. Smaller numbers are better.
<table>
<thead>
<tr>
<th>Example</th>
<th>Bounds</th>
<th>Tr. Lns.</th>
<th># iterations</th>
<th>Total (s)</th>
<th>Phase (s)</th>
<th># ∨</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARBAC</td>
<td>3 URP</td>
<td>5</td>
<td>13 4383</td>
<td>2 63</td>
<td>1 1 1 1</td>
<td>6 2</td>
</tr>
<tr>
<td>ARBAC</td>
<td>4 URP</td>
<td>5</td>
<td>15 2452</td>
<td>6 127</td>
<td>1 1 4 9</td>
<td>2</td>
</tr>
<tr>
<td>ARBAC</td>
<td>5 URP</td>
<td>5</td>
<td>27 2784</td>
<td>19 484</td>
<td>1 1 14 12</td>
<td>3</td>
</tr>
<tr>
<td>ARBAC</td>
<td>6 URP</td>
<td>5</td>
<td>37 3574</td>
<td>47 &gt;1800</td>
<td>1 3 1 38 12</td>
<td>4 4</td>
</tr>
<tr>
<td>ARBAC</td>
<td>7 URP</td>
<td>5</td>
<td>16 1609</td>
<td>42 &gt;1800</td>
<td>7 2 1 32 15</td>
<td>3</td>
</tr>
<tr>
<td>ARBAC</td>
<td>8 URP</td>
<td>5</td>
<td>154 670</td>
<td>769 &gt;1800</td>
<td>1 31 9 695 21</td>
<td>8</td>
</tr>
<tr>
<td>ARBAC</td>
<td>3 URP</td>
<td>10</td>
<td>45 69</td>
<td>13 4</td>
<td>1 2 1 9</td>
<td>4</td>
</tr>
<tr>
<td>ARBAC</td>
<td>4 URP</td>
<td>10</td>
<td>22 356</td>
<td>14 42</td>
<td>1 1 10 3</td>
<td></td>
</tr>
<tr>
<td>ARBAC</td>
<td>5 URP</td>
<td>10</td>
<td>35 4598</td>
<td>51 &gt;1800</td>
<td>1 4 38 16</td>
<td>5</td>
</tr>
<tr>
<td>ARBAC</td>
<td>6 URP</td>
<td>10</td>
<td>57 234</td>
<td>169 274</td>
<td>1 11 3 138 21</td>
<td>10</td>
</tr>
<tr>
<td>ARBAC</td>
<td>7 URP</td>
<td>10</td>
<td>68 551</td>
<td>454 &gt;1800</td>
<td>1 23 5 389 31</td>
<td>15</td>
</tr>
<tr>
<td>ARBAC</td>
<td>8 URP</td>
<td>10</td>
<td>213 274</td>
<td>1576 &gt;1800</td>
<td>1 96 24 1350 22</td>
<td>9</td>
</tr>
<tr>
<td>Unsat</td>
<td>3 URP</td>
<td>5</td>
<td>314 62753</td>
<td>73 &gt;1800</td>
<td>1 5 2 62 8</td>
<td>2</td>
</tr>
<tr>
<td>Unsat</td>
<td>4 URP</td>
<td>5</td>
<td>2610 22943</td>
<td>1800 &gt;1800</td>
<td>1 89 24 1626 10</td>
<td>2</td>
</tr>
<tr>
<td>Unsat</td>
<td>5 URP</td>
<td>5</td>
<td>1149 7151</td>
<td>&gt;1800 &gt;1800</td>
<td>1 76 21 1660 12</td>
<td>3</td>
</tr>
<tr>
<td>Unsat</td>
<td>6 URP</td>
<td>5</td>
<td>563 2423</td>
<td>&gt;1800 &gt;1800</td>
<td>1 63 20 1685 14</td>
<td>4</td>
</tr>
<tr>
<td>Unsat</td>
<td>3 URP</td>
<td>10</td>
<td>475 39580</td>
<td>221 &gt;1800</td>
<td>1 18 4 184 8</td>
<td>3</td>
</tr>
<tr>
<td>Unsat</td>
<td>4 URP</td>
<td>10</td>
<td>1283 10704</td>
<td>&gt;1800 &gt;1800</td>
<td>1 107 24 1593 11</td>
<td>4</td>
</tr>
<tr>
<td>Unsat</td>
<td>5 URP</td>
<td>10</td>
<td>538 2765</td>
<td>&gt;1800 &gt;1800</td>
<td>1 87 21 1639 14</td>
<td>4</td>
</tr>
<tr>
<td>Unsat</td>
<td>6 URP</td>
<td>10</td>
<td>291 702</td>
<td>&gt;1800 &gt;1800</td>
<td>1 83 20 1654 16</td>
<td>6</td>
</tr>
</tbody>
</table>

Fig. 2. Numeric Evaluation. Columns shaded gray give baseline performance in Alloy*. Timed-out experiments are indicated by “>1800”.

Sizes are aggregated over iterations as average and standard deviation. Higher bounds and trace lengths often lead to larger enabling causes and thus to larger learned constraints. There is a strong, although not universal, relationship between this metric and performance.

VII. RELATED WORK

Alloy* enables for-all-traces synthesis in a relational setting through higher-order universal quantification, which it achieves via CEGIS. While this is expressively sufficient, Sec. VII shows there is room for improvement. Modeling transition systems and checking temporal properties in Alloy are well studied (e.g., Giannakopoulos, et al. [14], DynAlloy [15], Vakili and Day [17], [18], and Electrum [19], [20]).

**Program Synthesis:** CEGIS was first introduced by Sketch [10], a seminal example of Syntax Guided Synthesis [5]. A representative sample of CEGIS in program synthesis includes program-repair hints [21], memory-consistency models [22], optimal synthesis [23] low-level bit-manipulation programs [24] and data structures [25], and concurrent programs subject to temporal safety goals [26]. Other algorithmic approaches include enumerative search [27], a QBF solver via a symbolic model-checker [28], and conflict-driven learning [29]. All of these focus on finding fixed programs, rather than initial configurations.

**Reactive Synthesis:** Reactive synthesis [6] synthesizes a system automaton satisfying given temporal properties. In contrast, our work takes properties and a system automaton as input and finds safe starting states. We are therefore able to search directly within the pre-established symbolic system.

Reactive synthesis for the GR(1) LTL fragment [30] has been useful for, e.g., robot controllers [31], [32]. Our properties are both more and less expressive; Alg. 1 only supports safety but permits operator nesting that GR(1) excludes.

**Configuration synthesis:** In the network-configuration domain, many works (e.g., ConfigAssure [33], Diekmann, et al. [34], [35], Zhang, et al. [36], NetGen [37], Merlin [38], FatTire [39], Propane [40], [41], and the NetComplete [42], [43] line of work) synthesize fixed configurations for specific domains such as firewalls, routing rules, BGP policies, and middleware configuration scripts. They handle properties involving connectivity, bandwidth, and other network-specific goals. In contrast, we consider synthesis within a domain-agnostic system model with online mutation. Interestingly, NetComplete can also autocomplete an existing partial configuration, reducing the search space by leveraging human expertise. This could be added to our implementation via (partial) exact Kodkod bounds during candidate generation.

**Parameter Synthesis:** Parametric transition systems allow a (fixed) initial configuration to affect system behavior. Parameter synthesis (e.g., [44], [45]) finds parameters under which the system meets desired properties. Once fixed, parameters do not change. Yet, some techniques from parameter synthesis are still applicable here; our approach is strongly related to Cimatti, et al. [46] who use an unbounded-trace model-checking algorithm to produce counterexamples. These works all accept state properties, which are weaker than general safety. Although it is possible to support arbitrary safety properties by adding additional state to the model, our approach works without any modifications to the state machine by identifying a proximal cause for failure which is then iteratively collapsed to produce blame in the initial state.

This work is partially supported by the US NSF, AFRL, and DARPA. We are grateful to Armando Solar-Lezama and James Bornholt for feedback at the beginning of this project, to Howard Reubenstein for discussions throughout, and to the anonymous reviewers for their remarks.