Synthesizing Mutable Configurations: Setting up Systems for Success

Tim Nelson
Brown University
tbn@cs.brown.edu

Natasha Danas
Brown University
ndanas@cs.brown.edu

Theophilos Giannakopoulos
Systems and Technology Research
theophilos.giannakopoulos@stresearch.com

Shriram Krishnamurthi
Brown University
sk@cs.brown.edu

***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 attacks.

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, *role*s mediate *user* access to *permission*s. 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: $ra : User \times Role$ and $pa : Role \times Permission$.

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:

- $canAssign : Role \times Role$ says which roles users with a given administrative role are empowered to assign.
- $require : Role \times Role \times Role$ positively filters role-assignment.
- $forbid : Role \times Role \times Role$ negatively filters role-assignment.

For someone to use role a to grant role r to user u, it must hold that (1) $(a, r) \in canAssign$; (2) $\forall\, r'$ such that $(a, r, r') \in require$, $(u, r') \in ra$; and (3) $\forall\, r'$ such that $(a, r, r') \in forbid$, $(u, r') \notin ra$.

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 synthesis 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.

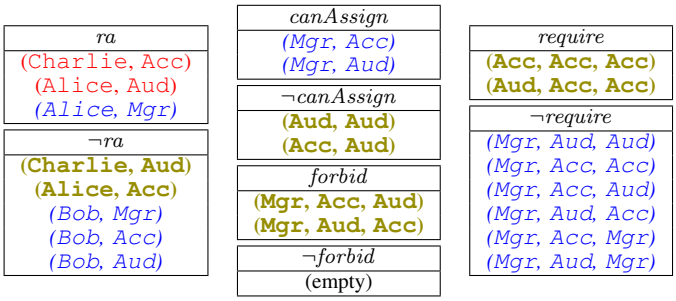

Fig. 1. Synthesized configuration. We typeset settings according to which goal they help ensure: (1): normal red; (2): **bold olive**; (3): *italic blue*. Raw output is edited for readability; role names are truncated, e.g., "`Accountant`" is "`Acc`". Negative settings, denoted with a ¬, are false in the configuration.

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 δ over a set of relations R and a set of R-properties Θ 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 Θ are LTL formalizations of the three goals in Sec. II.

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

The goal of synthesis is to provide a configuration (i.e., a start state of M) that satisfies Θ. For the Sec. II example, Fig. 1 gives a configuration that satisfies all 3 goals shown.

IV. Algorithmics

For simplicity, we consider the bounded-trace version of synthesis: given a machine M, a finite set of safety requirements Θ and a trace-length n, we seek an n-feasible configuration of the machine for Θ. Logically, this means satisfying $\exists c \forall \vec{s}\ .\ (s_0 = c \wedge \vec{s} \in \mathrm{Traces}_n(M)) \implies \Theta(\vec{s})$, where c ranges over configurations and $\vec{s}$ over traces, and s_0 denotes the first state in $\vec{s}$.

A. A First Attempt: Learning from counterexamples

Such problems often submit to the CounterExample-Guided Inductive Synthesis [10] (CEGIS) approach. The key idea is that, while the $\forall$ quantifier may be expensive to eliminate, *counterexamples* for Θ 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 $\exists c \forall \vec{s} \in S\ .\ (s_0 = c \wedge \vec{s} \in \mathrm{Traces}_n(M)) \implies \Theta(\vec{s})$ for *small (initially empty)* S; then **(2)** attempt to satisfy $\exists \vec{s}\ .\ s_0 = c \wedge \vec{s} \in \mathrm{Traces}_n(M) \wedge \neg\Theta(\vec{s})$. Should phase (2) yield a counterexample trace, it is added to S and phase (1) repeats. Otherwise, c is correct.

Adding a fixed trace $\vec{s}$ to S involves learning the instantiation of the quantifier-free subformula with the trace. Since $\neg\Theta(\vec{s})$ always holds, this means excluding the counterexample from the system's behavior. But $\mathrm{Traces}_n(M)$ 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 Θ, there will be a finite set of relational facts (possibly spanning multiple states in the trace) that suffice to violate Θ. 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 Θ. The revised algorithm, Alg. 1, 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 α_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 Θ. The only subtlety is that we must reframe the problem to return *traces* rather than a starting configuration. We use $\hat{M}$ to denote the formula that encodes $\mathrm{Traces}_n(M)$. 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.

```
1  L ← empty;
2  while true do
3      s_0 ← solve(α_0 ∧ L);
4      if unsat(s_0) then return fail();
5      candidate ← s_0;
6      s⃗ ← solve(candidate ∧ M̂ ∧ ¬Θ);
7      if unsat(s⃗) then return success(candidate);
8      w⃗ ← minimize(core(solve(s⃗ ∧ Θ)));
9      w⃗ ← buckets(filter(pc, in?(s⃗)));        // bucket by state
10     s⃗ · _ ← s⃗;                              // discard final state
11     while |w⃗| > 1 do
12         s⃗ · s ← s⃗ ;                          // isolate current prestate
13         w⃗ · w · w′ ← w⃗ ;                     // cause fragment in poststate
14         root ← solve(s ∧ ¬w′ ∧ δ) ;
15         φ ← filter(minimize(core(root)), in?(s));
16         w⃗ ← w⃗ · (w ∧ φ);                     // empty φ is treated as true
17     _ · w ← w⃗;       // isolate last remaining component of w⃗
18     L ← L ∧ ¬w;
```

Algorithm 1: Configuration Synthesis

• **Phase 3** (Line 8–Line 9): Determine a proximal cause. Given a counterexample trace $\vec{s}$, we seek a smallest justification for its failure on Θ. While the conjunction of all facts in $\vec{s}$ suffices, this is often much larger than needed. Consider the LTL formula Gp_{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 $\vec{s} \wedge \Theta$, where $\vec{s}$ is a conjunction of literals that fully describe the counterexample trace. The overall query must be unsatisfiable, or else $\vec{s}$ 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 $\vec{s}$ and sorting its components by state yields a vector of formulas $\vec{w}$ 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 $\vec{w}$ 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 Θ, 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 $\vec{w}$ is preserved.)

If the post-filter cause ϕ is empty, this means that there is no configuration that satisfies Θ. 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 an order-of-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 [12] 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):** Where are the performance bottlenecks of Alg. 1? We report on the example from Sec. II (**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: **s**ynthesis, **v**erification, **p**roximal cause, and **r**oot cause. Sub-1-second results are shown as 1 second to avoid potentially misleading zeros. Throughout, the vast majority of runtime is spent in **r**oot-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 #$\vee$ column reports how many disjuncts were in each iteration's learned constraint. Smaller numbers are better.

Example	Bounds	Tr. Ln.	# iterations		Total (s)		Phase (s)				# ∨	
			Alg. 1	Alloy*	Alg. 1	Alloy*	s	v	p	r	avg	sd
ARBAC	3 URP	5	13	4383	2	63	1	1	1	1	6	2
ARBAC	4 URP	5	15	2452	6	127	1	1	1	4	9	2
ARBAC	5 URP	5	27	2784	19	484	1	1	1	14	12	3
ARBAC	6 URP	5	37	3574	47	**>1800**	1	3	1	38	12	4
ARBAC	7 URP	5	16	1609	42	**>1800**	1	2	1	32	15	3
ARBAC	8 URP	5	154	670	769	**>1800**	1	31	9	695	21	8
ARBAC	3 URP	10	45	69	13	4	1	2	1	9	9	4
ARBAC	4 URP	10	22	356	14	42	1	1	1	10	10	3
ARBAC	5 URP	10	35	4598	51	**>1800**	1	4	1	38	16	5
ARBAC	6 URP	10	57	234	169	274	1	11	3	138	21	10
ARBAC	7 URP	10	68	551	454	**>1800**	1	23	5	389	31	15
ARBAC	8 URP	10	213	274	1576	**>1800**	1	96	24	1350	22	9
Unsat	3 URP	5	314	62753	73	**>1800**	1	5	2	62	8	2
Unsat	4 URP	5	2610	22943	**>1800**	**>1800**	1	89	24	1626	10	2
Unsat	5 URP	5	1149	7151	**>1800**	**>1800**	1	76	21	1660	12	3
Unsat	6 URP	5	563	2423	**>1800**	**>1800**	1	63	20	1685	14	4
Unsat	3 URP	10	475	39580	221	**>1800**	1	18	4	184	8	3
Unsat	4 URP	10	1283	10704	**>1800**	**>1800**	1	107	24	1593	11	4
Unsat	5 URP	10	538	2765	**>1800**	**>1800**	1	87	21	1639	14	4
Unsat	6 URP	10	291	702	**>1800**	**>1800**	1	83	20	1654	16	6

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* [13] 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. VI 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], [16], 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 middlebox 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]–[46]) 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.

References

[1] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar, "Why does the cloud stop computing?: Lessons from hundreds of service outages," in *ACM Symposium on Cloud Computing*, 2016.

[2] Amazon Web Services, "Summary of the Amazon S3 Service Disruption in the Northern Virginia (US-EAST-1) Region," https://aws.amazon.com/message/41926/, 2017, accessed March 31, 2019.

[3] Google official blog, posted by Ben Treynor, VP Engineering , "Today's outage for several Google services," https://googleblog.blogspot.com/2014/01/todays-outage-for-several-google.html, 2014, accessed March 31, 2019.

[4] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking Control of the Enterprise," in *Conference on Communications Architectures, Protocols and Applications (SIGCOMM)*, 2007.

[5] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *Formal Methods in Computer-Aided Design*, 2013.

[6] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *Principles of Programming Languages (POPL)*, 1989.

[7] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models," *Computer*, vol. 29, no. 2, pp. 38–47, 1996.

[8] R. Sandhu, V. Bhamidipati, and Q. Munawer, "The ARBAC97 model for role-based administration of roles," *ACM Trans. Inf. Syst. Secur.*, vol. 2, no. 1, Feb. 1999.

[9] R. Sandhu and Q. Munawer, "The ARBAC99 model for administration of roles," in *Proceedings of the 15th Annual Computer Security Applications Conference*, ser. ACSAC '99, 1999.

[10] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[11] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2007, pp. 632–647.

[12] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Theory and Applications of Satisfiability Testing*, 2003.

[13] A. Milicevic, J. P. Near, E. Kang, and D. Jackson, "Alloy*: A general-purpose higher-order relational constraint solver," in *International Conference on Software Engineering*, 2015.

[14] T. Giannakopoulos, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "Towards an operational semantics for Alloy," in *International Symposium on Formal Methods (FM)*, 2009.

[15] M. F. Frias, J. P. Galeotti, C. G. López Pombo, and N. M. Aguirre, "DynAlloy: Upgrading Alloy with actions," in *International Conference on Software Engineering*, 2005.

[16] G. Regis, C. Cornejo, S. Gutiérrez Brida, M. Politano, F. Raverta, P. Ponzio, N. Aguirre, J. P. Galeotti, and M. Frias, "DynAlloy Analyzer: A tool for the specification and analysis of Alloy models with dynamic behaviour," in *Foundations of Software Engineering*, 2017.

[17] A. Vakili and N. A. Day, "Temporal logic model checking in Alloy," in *International Conference on Abstract State Machines, Alloy, B, and Z*, 2012, pp. 150–163.

[18] A. Vakili, "Temporal logic model checking as automated theorem proving," Ph.D. dissertation, University of Waterloo, Ontario, Canada, 2016.

[19] J. Brunel, D. Chemouil, A. Cunha, T. Hujsa, N. Macedo, and J. Tawa, "Proposition of an action layer for Electrum," in *International Conference on Abstract State Machines, Alloy, B, and Z*, 2018.

[20] N. Macedo, J. Brunel, D. Chemouil, A. Cunha, and D. Kuperberg, "Lightweight specification and analysis of dynamic systems with rich configurations," in *Foundations of Software Engineering*, 2016.

[21] P. M. Phothilimthana and S. Sridhara, "High-coverage hint generation for massive courses: Do automated hints help CS1 students?" in *Conference on Innovation and Technology in Computer Science Education*, 2017.

[22] J. Bornholt and E. Torlak, "Synthesizing memory models from framework sketches and litmus tests," in *Programming Language Design and Implementation (PLDI)*, 2017.

[23] J. Bornholt, E. Torlak, D. Grossman, and L. Ceze, "Optimizing synthesis with metasketches," in *Principles of Programming Languages (POPL)*, 2016.

[24] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, "Synthesis of loop-free programs," in *Programming Language Design and Implementation (PLDI)*, 2011.

[25] C. Loncaric, M. D. Ernst, and E. Torlak, "Generalized data structure synthesis," in *International Conference on Software Engineering*, 2018.

[26] A. Solar-Lezama, C. G. Jones, and R. Bodík, "Sketching concurrent data structures," in *Programming Language Design and Implementation (PLDI)*, 2008.

[27] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur, "TRANSIT: Specifying protocols with concolic snippets," in *Programming Language Design and Implementation (PLDI)*, 2013.

[28] A. Gascón and A. Tiwari, "A synthesized algorithm for interactive consistency," in *NASA Formal Methods*, 2014.

[29] Y. Feng, R. Martins, O. Bastani, and I. Dillig, "Program synthesis using conflict-driven learning," in *Programming Language Design and Implementation (PLDI)*, 2018.

[30] N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of Reactive(1) designs," in *Verification, Model Checking, and Abstract Interpretation*, 2006.

[31] S. Maoz and J. O. Ringert, "GR(1) synthesis for LTL specification patterns," in *Foundations of Software Engineering*, 2015.

[32] ——, "On the software engineering challenges of applying reactive synthesis to robotics," in *Workshop on Robotics Software Engineering*, 2018.

[33] S. Narain, G. Levin, S. Malik, and V. Kaul, "Declarative infrastructure configuration synthesis and debugging," *J. Netw. Syst. Manage.*, vol. 16, no. 3, Sep. 2008.

[34] C. Diekmann, J. Naab, A. Korsten, and G. Carle, "Agile network access control in the container age," *IEEE Trans. Network and Service Management*, vol. 16, no. 1, pp. 41–55, 2019.

[35] C. Diekmann, S. Posselt, H. Niedermayer, H. Kinkelin, O. Hanka, and G. Carle, "Verifying security policies using host attributes," in *Formal Techniques for Distributed Objects, Components, and Systems*, 2014.

[36] S. Zhang, A. Mahmoud, S. Malik, and S. Narain, "Verification and Synthesis of Firewalls using SAT and QBF," *IEEE International Conference on Network Protocols (ICNP)*, 2012.

[37] S. Saha, S. Prabhu, and P. Madhusudan, "NetGen: Synthesizing data-plane configurations for network policies," in *Symposium on SDN Research (SOSR)*, 2015.

[38] R. Soulé, S. Basu, R. Kleinberg, E. G. Sirer, and N. Foster, "Managing the network with Merlin," in *Workshop on Hot Topics in Networks*, 2013.

[39] M. Reitblatt, M. Canini, A. Guha, and N. Foster, "FatTire: Declarative fault tolerance for software-defined networks," in *Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13, 2013.

[40] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker, "Don't mind the gap: Bridging network-wide objectives and device-level configurations," in *Conference on Communications Architectures, Protocols and Applications (SIGCOMM)*, 2016.

[41] ——, "Network configuration synthesis with abstract topologies," in *Programming Language Design and Implementation (PLDI)*, 2017.

[42] A. El-Hassany, P. Tsankov, L. Vanbever, and M. T. Vechev, "Network-wide configuration synthesis," in *International Conference on Computer Aided Verification*, 2017.

[43] ——, "NetComplete: Practical network-wide configuration synthesis with autocompletion," in *Networked Systems Design and Implementation*, 2018.

[44] F. Wang, "Symbolic parametric safety analysis of linear hybrid systems with BDD-like data-structures," in *International Conference on Computer Aided Verification*, 2004.

[45] G. Frehse, S. K. Jha, and B. H. Krogh, "A counterexample-guided approach to parameter synthesis for linear hybrid automata," in *International Workshop on Hybrid Systems: Computation and Control*, 2008.

[46] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, "Parameter synthesis with IC3," in *Formal Methods in Computer-Aided Design*, 2013.