Automated Combinatorial Test Generation for Alloy

Agustín Borda*†, Germán Regis*, Nazareno Aguirre*†, Marcelo Frias‡ and Pablo Ponzio*
*University of Rio Cuarto and CONICET, Rio Cuarto, Argentina
{aborda, gregis, naguirre, pponzio}@dc.exa.unrc.edu.ar
†Guangdong Technion-Israel Institute of Technology, Shantou, China
‡University of Texas at El Paso, El Paso, USA
mfrias4@utep.edu

***Abstract*—Specifications are an essential component of software development, and getting specifications right, especially *formal specifications*, can be very challenging. While the use of tools such as model finders and model checkers can be very effective for specification analysis through property checking, researchers have also realized that by the explicit provision of wanted and unwanted specification scenarios, in the style of testing in programs, specification assessment can be significantly enhanced. Thus, various testing and test generation techniques have been recently proposed for assessing formal specifications.**

In this paper, we present such a specification testing approach, in the form of a novel combinatorial testing technique for Alloy specifications, called COMBA. COMBA implements an automated partitioning of the state space of Alloy specifications solely based on elements of the specification (thus not requiring user intervention), and defines a family of test criteria, that indicate how such partitions are to be covered. The coverage of the partitions is defined by a family of combinatorial criteria that, given a positive integer t, require to cover through test cases all feasible t-uples of elements from different partitions. Finally, COMBA introduces an efficient algorithm to generate test cases that satisfy the combinatorial criteria. By leveraging on incremental SAT solving techniques, COMBA achieves significantly better performance in test generation.

We experimentally assess COMBA against existing test generation approaches for Alloy, using a large number of specifications with known errors. The results show that COMBA(with $t = 2$) runs faster, produces smaller test suites, and finds a significantly larger number of real bugs than related approaches.

***Index Terms*—Alloy, Specification Testing, Combinatorial Testing, Incremental SAT Solving**

I. Introduction

Specification is an essential component of software design and development. Writing specifications that correctly capture the problem domain as well as the characteristics of the system-to-be can be very challenging, especially when formal languages are used to capture software specifications. Thus, developers often resort to different analysis tools such as model checkers and model finders, to help in the construction of specifications. These tools are particularly useful for debugging, analyzing, and improving formal specifications, as they enable developers to identify misconceptions, incomplete as well as inconsistent assumptions, and in general design flaws in the characteristics of the specified system, early in the development process, thus leading to more robust and reliable software systems [6].

Alloy is a widely used formal specification language that has been successfully employed in different software engineering activities, including the analysis of software designs [12], [13], the analysis of communication protocols [48], as well as a backend for test case generation [31], [36] and software verification [8], [10], among other activities. One of Alloy's main advantages –that contributed to its success– is that its specifications are amenable to automated analysis via software tools, the most prominent being the Alloy Analyzer. The Alloy Analyzer implements two kinds of automated analyses of specifications. First, it allows one to run *simulations* of a specification, i.e., it finds (bounded) instances satisfying a given specification (if such instances exist), and provides these to the user. Second, it allows one to perform a bounded verification of properties of the specification, and generates and provides counterexamples, if these exist. The analysis implemented by the Alloy Analyzer relies on SAT solving, and requires the user to provide bounds on the data domains of the specification. The Analyzer translates the bounded specification and the property to be analyzed to a propositional formula, that is then fed to a SAT solver to search for satisfying instances or counterexamples.

Although bounded simulation and property verification are of great help to the user, especially for assessing the correctness of a specification, by themselves these approaches are often not enough. Formal specifications are typically affected by issues such as overspecification defects, where the specification is overconstrained and discards valid instances, and underspecification defects, where the specification accepts invalid instances. Overspecification defects lead to considering invalid properties as valid properties, in cases where some of the discarded valid instances are exactly the ones that would violate the property. Overspecification defects are not easily found by simulation, as they are only detected by the lack of specific instances. As the user often explores only a few instances, he might miss the ones that reveal the error. Underspecification issues might not be found for the same reason: the instances witnessing underspecification might not be presented to the user during manual exploration.

To address these problems, automated test generation approaches for Alloy specifications have been proposed, in an effort to provide the user with mechanisms to gain confidence about the correctness of the specifications [41], [43], [40], [17]. Two main testing approaches exist for Alloy. AUnit proposes a systematic way to cover subformulas of a specification with test cases [40], and it resembles code coverage criteria in

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00240

traditional software testing (line coverage, branch coverage) [5]. MuAlloy is a mutation based approach, it creates small syntactical modifications of the specification (that simulate typical specification errors made by engineers), and generates test cases to "kill the mutants" [43], [17], in a similar way to mutation testing for software[5].

Notice that, resembling traditional software testing, systematic testing approaches for Alloy define *coverage criteria* over specifications, where each criteria defines a set of *test requirements* (subformulas for AUnit, mutants for MuAlloy) that must be covered with specific test cases [5]. The goal of coverage criteria is the same as in software testing: a good coverage criteria should guide the testing process (manual or automated) to produce a relatively small set of test cases that find most of the flaws in the specification. In the case of formal specifications, the flaws are in many cases related to underspecification and overspecification issues, as discussed above. These are examples of the kind of errors that testing approaches for Alloy aim to discover.

In this paper, we introduce a novel automated testing approach for Alloy specifications, called COMBA, inspired by the combinatorial testing approaches for software systems [5]. The first stage of COMBA consists of an automated approach to partition the state space of the Alloy specification under test. The partitioning is based on a combination of structural aspects of the specification, as captured by the specification's signatures and fields, and more semantic-related characteristics, as captured by the user through the predicates and assertions of the specification. More precisely, the partitioning considers test requirements that involve: covering signatures and fields by their cardinality, with goals for covering these each field/signature with zero or more elements. Additionally, predicates and assertions present in the specification lead to equivalence classes for covering these properties with their satisfaction and non-satisfaction (i.e., covering these properties making them true and false, respectively). The rationale here is that this partitioning enforces more diversity in "semantic" equivalence classes (based on predicates and assertions), by combining these with "syntactic" equivalence classes (based on signatures and fields). Note that, the coarseness of the cardinality based partitions allows us to maintain a manageable number of combinations. Using this approach, COMBA partitions the state space of Alloy specifications without requiring user intervention.

COMBA defines a family of test adequacy criteria that indicates how the partitions of the previous stage should be covered with tests cases. Given a positive integer `t`, the criteria require to cover all feasible `t`-uples of equivalence classes from different partitions with test cases. In a second stage, COMBA generates all the test requirements to be covered (sets of `t`-uples). We call COMBA_t the approach that partitions the state space of the specification under test (first stage), generates the `t`-uples that conform the test requirements (second stage), and then creates test cases to cover all the `t`-uples (third stage).

In this way, the third stage of COMBA is an algorithm that efficiently generates tests that satisfy COMBA's combinatorial adequacy criteria. Since the number of `t`-uples to be covered by COMBA_t is usually very large, test generation can be very expensive. To tackle this issue, the test generation algorithm of COMBA is designed to exploit incremental SAT solving for efficient generation [9], [28]. The algorithm works at the level of propositional CNF formulas [9], that result from translating the (bounded) Alloy specification to a SAT problem. The algorithm is designed to leverage the mechanisms of modern incremental SAT solvers to iteratively query the solver for test cases to cover all test requirements, one after another. It achieves this without having to generate a new propositional formula or restart the solver for each new test requirement to be covered, as it would be the case using a non-incremental solving approach. The use of incremental SAT solving has been shown very effective for improving the efficiency of approaches that need to make a large number of queries to a SAT solver (e.g. [3], [32]).

We experimentally assess COMBA against state-of-the-art (SOTA) test generation approaches for Alloy, AUnit, [40], MuAlloy [43], [17], and Scenario Tour [37] using 18 case studies taken from the literature with 29.946 known real bugs [27], [26]. The results show that COMBA_2 is faster, generates smaller test suites, and it finds a significantly higher number of real bugs than SOTA approaches. In addition, the use of incremental SAT solving make COMBA_1 and COMBA_2 faster by 6.5 and 11.1 times, respectively, w.r.t. versions of the approaches that use traditional SAT solving.

To summarize, the contributions of this paper are:

- A combinatorial test generation approach for Alloy, COMBA, that automatically partitions the state space of the specification, and defines a family of combinatorial criteria to create test requirements from the partitions.
- An efficient algorithm to generate test suites for the combinatorial criteria, based on incremental SAT solving; a crucial part of COMBA's approach.
- A comprehensive experimental assessment systematic testing approaches for Alloy, using a benchmark consisting of 18 case studies, with a total of 29.946 known real bugs. The results of the assessment show that COMBA_2 is faster, generates smaller test suites, and it finds a significantly larger number of real bugs than SOTA approaches.

II. Background

A. The Alloy Specification Language

Alloy is a formal specification language, that proposes to capture software properties using a logical language with a relational flavor [11], [12]. This language follows the specification approach of model-oriented specification languages [45], [4], [15], which capture software and problem domains in terms of data domains, their properties, and transformations between these domains. But as opposed to previous languages, from its inception, Alloy has put a strong emphasis on automated analysis [14]. Indeed, the language is accompanied by an efficient, fully automated, mechanism for bounded analysis

of specification properties. This analysis mechanism resorts to SAT solving to query for the bounded satisfiability or bounded validity of Alloy formulas [12], [13].

Alloy features a simple syntax and semantics for specifications. Moreover, the syntactic elements in Alloy specifications have an intuitive interpretation close to object-oriented abstractions, which together with the relational nature of the language, makes it accessible to software developers [12], [13]. Alloy allows for the definition of data domains via the concept of *signature*. Signatures can have *fields*, whose associated types are relations, of any (positive) arity, between signatures. Alloy specifications can be equipped with formulas written in *relational logic*, the logic underlying Alloy. Relational logic is a first-order logic extended with relational operators, such as union (+), intersection (&), join (.), and most importantly, transitive closure (^). Alloy's logic is therefore strictly more expressive than first-order logic [12]. Alloy formulas use standard logical connectives (`and`, `or`, `not`, etc.), quantifiers (`all`, `some`, `no`, the latter corresponding to "there is no"), relational inclusion (`in`) and equality (=). *Predicates* are parameterized formulas, that can be used to state properties, and capture operations, among other things. *Assertions* are used to state *intended* properties, i.e., formulas that are expected to hold. Both predicates and assertions can be automatically analyzed, by searching for satisfying instances, in the case of predicates, and for counterexamples, in the case of assertions.

Starting with version 6, Alloy was extended to incorporate linear temporal logic (LTL) constructs, in an effort to facilitate the specification of dynamic properties of systems [25]. The keyword `var` allows to specify mutable *signatures* and *fields*, that is, relations that can change over time. Instances of such relations are *traces*, with at most `N` time steps, where `N` is a bound on the maximum length of traces that must be provided by the user. Temporal properties over traces can be specified using the following operators. `'` denotes the value of a mutable relation in the next time step. `after`, `eventually`, `always` are the typical LTL operators, that specify that a property is valid in the next, in some, and in all time steps, respectively [25].

B. Incremental SAT solving

The test generation algorithm of COMBA is designed to exploit the "solve with assumptions" mechanism of modern SAT solvers [9], [28]. Basically, solve with assumptions allows one to query the solver to find an instance satisfying the current formula under the assumption that some variables $v_1, v_2, .., v_k$ are set to true (the solver allows to set some variables to false as well [9]). The idea is that the assumptions $v_1, v_2, .., v_k$ are only valid for the current query, the solver discards the assumptions after it answers the query (no matter the result). That is, the solver stays "clean" after each query with assumptions, and it can be queried again afterwards using a different set of assumptions. To abstract away the implementation details, we assume a `Solver` class, with a `solve_assumptions([vars])` method that takes a list of variables as assumptions (the variables assumed to be true).

```
1  abstract sig Source {}
2  sig User extends Source {
3    profile: set Work,
4    visible: set Work
5  }
6  sig Institution extends Source {}
7
8  sig Id {}
9  sig Work {
10   ids: some Id,
11   source: one Source
12 }
13 // [...]
14 pred owner[] {
15   all u:User, w:Work| w in u.profile implies
16 // Defect:
17     (some i:Institution| u in w.source or i in w.source)
18 // Fix:
19 //(u in w.source or some i:Institution| i in w.source)
20 }
21 // [...]
```

Fig. 1. Excerpt of a defective version of `cv` from Alloy4Fun

```
1  pred instance21[] {
2    some disj Id0 : Id {
3    some disj Work0: Work {
4    some disj Source0 : User {
5      no Institution
6      Id = Id0
7      Work = Work0
8      User = Source0
9      no visible
10     ids = Work0 -> Id0
11     profile = Source0 -> Work0
12     source = Work0 -> Source0 }}}
13 }
14
15 run { instance21[] and owner[] } for 5
```

Fig. 2. A failing test with an instance generated by COMBA_2

In Section IV-B, we discuss how test requirements generated by COMBA can be defined as tuples of propositional variables of the CNF formula that encodes the (bounded) Alloy as a SAT problem. Hence, we can employ `solve_assumptions` to obtain test suites that satisfy COMBA's combinatorial criteria, by querying the solver one test goal after another, without having to generate a new CNF formula for each test goal, and without having to restart the solver each time. Not as obvious yet still crucial for the performance of COMBA, the learning mechanisms of the incremental solvers speed up subsequent invocations by learning conflict clauses from past invocations [9], [28]. In this way, the use of incremental SAT solving allows COMBA to make thousands of queries to the solver and yet perform very well.

Finally, we would like to remark that the typical incremental SAT solving mechanism used to enumerate instances in the Alloy Analyzer does not work for our purposes. This mechanism consists of incrementally adding *clauses* to the CNF formula. However, clauses added using this approach are kept for all the subsequent queries [9]. For example, a clause added for covering test requirement v_1, v_2 would interfere with a subsequent clause added to cover a different one, say v_2, v_3 (v_1 would still be required at this point).

III. Illustrative example

In this section, we illustrate COMBA by means of an example. We show how COMBA generates an error revealing instance for (a buggy variant of) the Alloy4Fun `cv` specification [27]. `cv` models a system that keeps track of the works in the curriculum vitae of users. Signature `Work` defines the set of works in the system, and `Source` the set of feasible sources for works. Sources can be either users or institutions, as `Source` is an abstract signature extended by signatures `User` and `Institution`. Each work has at least one `Id` (due to the `some` modifier), as defined by the relation `ids` of `Work`. Also, each work has exactly one `Source` (due to the `one` modifier), defined by relation `source` of `Work`, which as stated before can be either an `User` or an `Institution`.

Predicate `owner` (lines 14-20 in Fig. 1) should formally describe the property: "A user profile can only have works added by himself or some external institution". The property is correctly described by the following Alloy formula. The defect in the specification is in line 17 of Fig. 1. The buggy `owner` predicate rejects cases where there are no institutions in the model, since such cases make the consequent of the implication false (the existential quantifier `some` becomes false when the set `Institution` is empty). This clearly does not capture the intended behavior for `owner`.

The following Figure shows the relevant partitions generated by COMBA for this specification:

$\pi_{Institution}$ =
- v_1 = no Institution,
- v_2 = some Institution}

π_{owner} =
- v_3 = owner[],
- v_4 = not owner[]}

COMBA generates a partition $\pi_{Institution}$ for the `Institution` signature with two equivalence classes: the first describes the instances where there are no institutions (the Alloy formula `no Institution` holds), and the second the instances where there is at least one institution (formula `some Institution` holds). For each partition, COMBA instruments the Alloy specification by introducing a unique propositional variable that is true if and only if the corresponding formula holds. We denote this by v_i = `pred` in the definition of the partitions, where `pred` is an Alloy predicate. Thus, by setting v_i to true, COMBA forces the solver to generate instances that belong to the corresponding partition. For example, v_1 is true exactly for instances where there is no institution (`no Institution` holds). COMBA also creates a partition using the `owner` predicate, that contains two equivalence classes, `owner[]` and `not owner[]`. That is, it splits the state space into instances where `owner` holds and where it does not, respectively. COMBA creates partitions in a similar way for the remaining signatures, relations (`ids`, `source`, etc), predicates and assertions; we omit them here for space reasons.

Given an integer $t > 0$, COMBA$_t$ creates test requirements in a combinatorial manner for the partitions, that is, it tries to cover all combinations of t-uples of different partitions with test cases. In our example, for $t = 2$, COMBA$_2$ produces the following test requirements:

```
TR = {
  (no Institution, owner[]),
  (no Institution, not owner[]),
  (some Institution, owner[]),
  (some Institution, not owner[])
}
```

```
TRV = {
  (v1,v3),
  (v1,v4),
  (v2,v3),
  (v2,v4)
}
```

Notice that test requirements can be expressed in terms of the Alloy predicates that must hold (`TR` at the left), or equivalently as the propositional variables that enforce the corresponding predicates to hold (`TRV` at the right part).

COMBA$_2$ generates an instance that reveals the error when it tries to cover test requirement `(no Institution, owner[])`. It produces the valuation shown in the test in Fig. 2. A test in Alloy is a pair `(i,c)`, where `i` is an instance and `c` is a command that indicates whether `i` is expected to be a valid or an invalid instance of the specification [41]. If the execution of the command fails (the command states that the instance should be valid but it is not, or vice versa), the test reveals an error in the specification. Lines 2-12 in Fig. 2 describe by extension the instance generated by COMBA$_2$. Line 15 shows the command of the test, which states that `owner` must evaluate to true in the instance. The command fails for the instance, as `owner` is not satisfied for and instance with no institutions (see line 5 in Fig. 2).

IV. COMBA: Combinatorial testing for Alloy

In this section, we describe COMBA in detail, which includes: (i) partitioning the state space of the specification (Sec. IV-A), (ii) the definition of test requirements (Sec. IV-B), and (iii) the generation of test cases to cover test requirements (Sec. IV-C).

A. Partitioning approach for Alloy specifications

```
1  fun make_partitions(Spec): Seq(Partition) {
2    res = []
3    for S in signatures(Spec)
4      v = new_var(); w = new_var();
5      res = res ++ [{ v = no S, w = some S }]
6    for R in fields(Spec)
7      v = new_var(); w = new_var();
8      res = res ++ [{ v = no R, w = some R }]
9    for U in functions(Spec)
10     v = new_var(); w = new_var();
11     res = res ++ [{ v = no U, w = some U }]
12   for P in predicates(Spec)
13     v = new_var(); w = new_var();
14     res = res ++ [{ v = P, w = not P }]
15   for A in asserts(Spec)
16     v = new_var(); w = new_var();
17     res = res ++ [{ v = A, w = not A }]
18   return res
19 }
```

Fig. 3. Partitioning approach for Alloy specifications

For the remaining of this section, let us assume a fixed Alloy specification `Spec`. Algorithm 3 shows a pseudocode of COMBA's approach to create partitions. As illustrated in Section III, COMBA creates a partition for each signature and each relational field of the specification (Lines 3-8 in Fig. 3). Both signatures and relational fields are represented

```
1  fun test_reqs(Spec, p: Seq(Partition), t: int):
2                                Set(Tuple(Var)) {
3    res = {}
4    for i1 in 1..len(p) {
5      for eq1 in 1..len(p[i1]) {
6        v1 = var(p[i1][eq1])
7        for i2 in (i1+1)..len(p) {
8          for eq2 in 1..len(p[i2]) {
9            v2 = var(p[i2][eq2])
10           ...
11           for it in (it−1+1)..len(p) {
12             for eqt in 1..len(p[it]) {
13               vt = var(p[it][eqt])
14               res = res ∪ {(v1,v2,..,vt)}
15           }}}}}}
16   return res
17 }
```

Fig. 4. Generation of test goals given a sequence of partitions

by relations in Alloy. Thus, the partitioning approach creates equivalence classes bases on their cardinality: one for the case the relation has no elements (described by formula `no S`), and another for when relation has at least one element (described by `some S`). As discussed in Section III, COMBA instruments the Alloy model with a new propositional variable to represent each equivalence class. We assume a method `new_var` that creates fresh variables each time is called. For example, in line 4 two new propositional variables `v` and `w` are created, such that `v` represents the equivalence class where S has no elements (`v = true` iff `no S`), and `w` is represents the equivalence class where S has at least one element (`w = true` iff `some S`). In general, each new propositional variable `v` created by `new_var` is used to represent exactly one equivalence class, by defining `v = pred`, where `pred` is an Alloy predicate that characterizes the class. In this way, setting `v` to true forces the solver to generate an instance that satisfies `pred`, i.e., an instance that belongs to the equivalence class represented by `v`. The test generation algorithm of COMBA heavily relies on this observation. Notice that, the instrumentation with new propositional variables is performed behind the scenes by COMBA. After test generation is finished, the user is presented with test cases that only refer to the syntactical elements of the original specification.

Similarly, COMBA creates partitions for functions based on the cardinality of their return value (Lines 9-11). COMBA also makes partitions using each predicate (see Section III) and assertion, by considering the cases where the predicate/assertion holds and does not hold (lines 12-17).

The partitioning scheme defined here supports the static and the dynamic specification styles (introduced in Alloy 6 [25]). To see why, consider that COMBA requires generating tests cases to make predicates true/false. For example, for the Alloy4Fun `trains_ltl` model, COMBA_1 will produce an instance to make the following predicate true: "it is always the case that the signal eventually becomes Green". In addition, for temporal models, we partition temporal signatures/relations as: (i) always empty, and (ii) eventually non-empty.

B. Definition of test requirements

Let $\texttt{p}= \pi_1, \pi_2, ..., \pi_n$ be the partitions induced in `Spec` by the `make_partitions` algorithm (Fig. 3). COMBA defines a family of coverage criteria. Given a positive integer `t` provided by the user, COMBA_t aims to cover every `t`-uple of equivalence classes from different partitions from `p`. We call such tuples the test requirements for COMBA. `test_reqs` in Figure 4 is a pseudocode of the algorithm used by COMBA to generate test requirements. `test_reqs` is a typical algorithm to compute the cartesian product of `t`-uples of equivalence classes from different partitions. It iterates over `t` different partitions, with indexes $\texttt{i}_1, \texttt{i}_2, .., \texttt{i}_t$ in `p`, draws equivalence classes with indexes $\texttt{eq}_1, \texttt{eq}_2, .., \texttt{eq}_t$ from each partition, and obtains the variables $\texttt{v}_1, \texttt{v}_2, .., \texttt{v}_t$ that represent the corresponding equivalence classes. Then, it adds test requirement $(\texttt{v}_1, \texttt{v}_2, .., \texttt{v}_t)$ to the result in line 14. Note that, the algorithm avoids creating tuples that represent the same test requirement in different order, e.g., $(\texttt{v}_1, \texttt{v}_3)$ and $(\texttt{v}_3, \texttt{v}_1)$. It does so by ensuring that $\texttt{i}_1 < \texttt{i}_2 < ... < \texttt{i}_t$ while iterating over the partitions in `p`.

We say test requirement $\texttt{tr}=(\texttt{v}_1, \texttt{v}_2, .., \texttt{v}_t)$ is covered by a instance `I` if `I` satisfies the facts of `Spec` and belongs to equivalence classes $\texttt{eq}_1$, $\texttt{eq}_2$, and $\texttt{eq}_t$. In addition, we say that test requirement `tr` is infeasible when there is no valid instance (satisfying the facts) of `Spec` that covers `tr`. The occurrence of infeasible test goals is typical when using coverage criteria for software testing [5]. In COMBA, infeasible test goals can appear due to the facts of the specification making some equivalence classes unsatisfiable, or due to some combinations of equivalence classes being impossible to satisfy when taken in conjunction (and/or in conjunction with the facts). When COMBA cannot produce an instance to cover an infeasible test requirement `tr`, COMBA discards `tr` and continues with the next.

C. Combinatorial test generation approach

```
1  fun make_neg_partitions(Spec):(Spec, Seq(Partition)){
2    res = []
3    Spec', facts = remove_facts(Spec)
4    NF = "fact { or{not F | for F in facts} }"
5    // The disjunction of the negated facts is the
6    // only fact in Spec'
7    add_fact(Spec', NF)
8    for F in facts {
9      let F = "fact F { body }"
10     let PF = "pred PF { body }"
11     // PF is used to ask for instances that
12     // satisfy/do not satisfy the original F
13     add_pred(Spec', PF)
14     v = new_var(); w = new_var();
15     res = res ++ [{ v = PF, w = not PF }]
16   }
17   for S in signatures(Spec')
18     v = new_var(); w = new_var();
19     res = res ++ [{ v = no S, w = some S }]
20   for R in fields(Spec')
21     v = new_var(); w = new_var();
22     res = res ++ [{ v = no R, w = some R }]
23   return (Spec', res)
24 }
```

Fig. 5. Partitioning approach for the generation of negative instances

1) Generation of negative instances: As usual in software testing [5], we want to have some test cases with invalid instances, that check that the instances are rejected by the specification. We call these negative tests. Negative instances are useful, for example, to find overspecification errors, since they represent instances that are currently rejected by the specification, but this might be due to overly restrictive facts.

COMBA's approach to create partitions for the generation of negative instances is shown in Figure 5. `make_neg_partitions` first needs to and create a new specification `Spec'` by removing the original facts of `Spec` (line 4). Then, it adds the disjunction of the negations of the original facts as a fact for `Spec'` (line 7). The reason is that we want the valid instances of `Spec'` to be the invalid instances of `Spec` (and vice versa). Thus, the added fact forces the valid instances of `Spec'` to violate at least one of the original facts.

Then, the algorithm iterates over the original facts (lines 9-17), and creates for each fact `F` a new predicate `PF` with the same body as `F` (lines 9-11). Predicates `PF` are added to `Spec'` (line 14). Then, for each `PF` the algorithm creates a partition as if it were a normal predicate, i.e., it will make test generation cover the partition by making the predicate true and false (lines 15-16). In this way, covering the equivalence classes results in tests that satisfy and violate the original fact `F`, respectively. After generating test requirements for falsifying the original facts, we add additional partitions to ensure that COMBA tests the negation of the facts with a few different instances (lines 18-23). Thus, we partition the signatures and relational fields of `Spec'` in the same way as we did in `make_partitions` (Fig. 3).

2) COMBA's test generation approach: A pseudocode of COMBA's test generation approach is shown in Figure 6. `gen_tests` takes the Alloy specification under test, `Spec`, the scopes for test generation, `scope`, and a positive integer `t` to choose the combinatorial criteria to be employed. As typical in Alloy, `scope` defines the maximum number of allowed elements for each signature, and the maximum length of traces for temporal specifications (if needed). As output `gen_tests` yields a test suite that covers all feasible `t`-uples for the partitions induced in `Spec`.

`gen_tests` first generates happy path tests (lines 4-11). It invokes `make_partitions` to create a sequence of partitions `p` for `Spec` (line 4). We assume a `Translation` class that returns a translator object `tr`, which abstracts away all the low level details of the translation from Alloy to a CNF formula using the given scopes (line 5). The translation is performed by method `translate`, which yields a CNF formula (line 6). Then, `gen_tests` invokes routine `gen_instances` (defined in lines 24-38), which is in charge of producing a set of positive instances to achieve combinatorial coverage of `t`-uples for the partitions in `p` (line 7).

`gen_instances` first initializes the incremental SAT solver using the provided CNF formula (line 25). It initializes a set `covered` to store the covered test requirements during the generation of instances (line 27). Then, it iterates over the

```
1  fun gen_tests(Spec, scope: int, t: int): Test Suite {
2    suite = []
3    // Generation of positive tests
4    p = make_partitions(Spec)
5    tr = Translation(Spec, scope)
6    CNF = translate(tr, Spec)
7    posI = gen_instances(CNF, p, t)
8    for in posI {
9      test = gen_alloy_test(Spec, scope, I, tr, true)
10     suite = suite ++ [test]
11   }
12   // Generation of negative tests
13   Spec', p' = make_neg_partitions(Spec)
14   tr' = Translation(Spec', scope)
15   CNF' = translate(tr', Spec')
16   negI = gen_instances(CNF', p', t)
17   for I in negI {
18     test = gen_alloy_test(Spec', scope, I, tr', false)
19     suite = suite ++ [test]
20   }
21   return suite
22 }
23
24 fun gen_instances(CNF, p: Seq(Partition), t: int) {
25   solver = Solver(CNF)
26   res = []
27   covered = {}
28   for (v1,v2,..,vt) in test_reqs(p, t) {
29     if (v1,v2,..,vt) in covered continue
30     if solve_assumptions(solver,[v1,v2,..,vt])=SAT {
31       I = get_instance(solver)
32       res = res ++ [I]
33       // Mark all test reqs satisfied by I as covered
34       covered = covered ∪ covered_tuples(I)
35     }
36   }
37   return instances
38 }
39
40 fun gen_alloy_test(Spec, scope: int, I: Instance,
41     tr: Translation, pos: bool): Test {
42   predI = gen_alloy_pred(tr, I)
43   let predI = "pred I() { body }"
44   if (pos) { // Positive test, SAT in Spec
45     assert = {}
46     for P in (predicates(Spec) ∪ asserts(Spec))
47       if (eval(tr, "I and P")))
48         assert = assert ∪ {P}
49       else
50         assert = assert ∪ {not P}
51     A = and{a | a in assert}
52     return ("pred I() { body }",
53         "run { I and A } for scope expect 1")
54   }
55   else // Negative test, UNSAT in Spec
56     return ("pred I() { body }",
57           "run { I } for scope expect 0")
58 }
```

Fig. 6. COMBA's test generation algorithm

test requirements yielded by function `test_reqs` (Fig. 4) for the current partitions `p` and the given `t` (line 28). Recall that test requirements are tuples of variables : $(v_1, v_2, .., v_t)$, which represent equivalence classes from different partitions of `p`. If the current test requirement has been already covered by a test, `gen_instances` moves to the next one (line 30). Otherwise, it queries the SAT solver under the assumptions that variables $v_1, v_2, .., v_t$ must be set to true, which is equivalent to asking the solver for an instance that covers the current test requirement (line 31). If the SAT solver finds such an instance (the answer to `solve_assumptions` returns SAT), the algorithm stores the instance in `res` (line 32).

The instance I found by the solver is an assignment of truth values to propositional variables in the CNF formula. Notice that, I must assign truth values to variables that represent equivalence classes, i.e., to the variables created by new_var when make_partitions is invoked (see Figure 3). Function covered_tuples (line 34) generates all the t-uples of variables that represent equivalence classes and are set to true in I. These are the test requirements covered by I. That is, a single instance might cover many test requirements.

For example, the instance in Figure 2 makes true the following variables that represent equivalent classes: v_1 (no Institution), v_3 (owner), v_6 (some User), v_8 (some profile), v_9 (no visible), v_{12} (some Id), v_{14} (some Work), v_{16} (some ids), v_{18} (some source). Thus, for t=2, covered_tuples(I) (line 34) creates all pairs of the aforementioned variables, that are covered by the I:

```
{ (v1, v3), (v1, v6), (v1, v8), (v1, v9), (v1, v12),
  (v1, v14), (v1, v16), (v1, v18), (v3, v6), (v3, v8),
  (v3, v9), (v3, v12), (v3, v14), (v3, v16), (v3, v18), ...}
```

Afterwards, the tuples produced by covered_tuples are stored in set covered (line 34) (i.e., are considered covered by the algorithm). At the end, a set of instances that achieve combinatorial coverage of t-uples is returned (line 37).

gen_tests then creates an Alloy test for each positive instance yielded by gen_instances (lines 8-11). To achieve this, gen_alloy_test is invoked for each instance I (line 9). gen_alloy_test (defined in lines 40-58) first converts the instance I, which at this point is an assignment of truth values to propositional variables, to an Alloy predicate (line 42). For this it employs a mapping from propositional variables to syntactical elements of the Alloy specification provided by the Alloy Analyzer [12] (which we assume is encapsulated in the translation object tr). We omit the implementation details, and simply invoke method gen_alloy_pred, which we assume it generates the corresponding predicate predI using I and tr. An example of such a predicate is shown in Figure 2. We are left with adding regression assertions to complete the command of the test. Thus, for each predicate and assertion P of Spec, the translator tr employs the Alloy evaluator (implemented in the Alloy Analyzer) to check whether the instance satisfies P or not P, and adds either P or !P to the list of predicates satisfied by the instance, assert (lines 45-50). Then, a single Alloy expression with the conjunction of all the assertions is generated (line 51). The resulting test consists of invoking the Alloy predicate generated for I, and a run command stating that I satisfies the assertions (lines 52-53). All the generated positive tests are stored in the resulting test suite, suite (line 10).

Afterwards, gen_tests starts the generation of negative instances (lines 13-20). It generates the "negative" specification Spec′, and partitions p′ for it, by invoking make_neg_partitions (line 13). Similarly to the generation of positive instances, gen_tests translates Spec′ to a new CNF formula, CNF′ (lines 14-15). Then, gen_instances is invoked again, but this time with the goal of covering the partitions in p′ (line 16). Negative instances must also be converted to tests in the Alloy language (lines 17-20). This is performed by calling gen_alloy_test for each negative instance (line 18). The yielded negative Alloy tests are then added to the resulting test suite, suite (line 19). For negative instances gen_alloy_test first generates an Alloy predicate, in the same way as for positive ones (lines 42-43). As negative instances are by definition rejected by the facts of the original Spec, the command of the test must ensure that running the test returns UNSAT. Hence, gen_alloy_test adds expect 0 to the command of negative tests (lines 56-57).

Finally, gen_tests terminates and returns the generated test suite, suite (line 21).

V. Experimental Evaluation

We now experimentally assess COMBA. Our goal is to answer the following research questions:

- RQ1: How efficient is COMBA compared to related techniques?
- RQ2: How many tests does COMBA generate in comparison with related techniques?
- RQ3: How effective is COMBA at finding bugs with respect to related approaches?
- RQ4: Does incremental SAT solving contribute to the performance of COMBA?
- RQ5: How many unfeasible test requirements are generated by COMBA?

A. Experimental Setup

To evaluate the approach, we use specifications from two sets of subjects, the latest version of Alloy4Fun [26], and three more complex specifications taken from the literature. Alloy4Fun is composed of various subjects, each consisting of a correct (reference) Alloy specification, and various faulty variants of the same specification. Each faulty variant is a syntactically valid Alloy specification containing semantic errors involuntarily introduced by a human (a student) while trying to capture the right specification. The benchmark includes both static (non-temporal) and temporal models. The static models (called **A4F Static**) comprise: **class_fol** and **class_rl**, models of a school classroom, in first order logic and relational logic, resp.; **graph**, a model of the graph data structure; **cv** and **cv2**, old and new versions of a Curriculum Vitae specification, resp.; **courses**, a model of courses taught in a school; **network**, a specification of the network connections in a social media app; **trash_fol** and **trash_rl**, file system trash can specification in first order logic and relational logic, resp.; **lts** a model of labeled transition systems; **trains**, a specification of a train station; and **prod** and **prod2**, two alternative specifications of production lines. The temporal models (called **A4F Temporal**) employ Alloy 6's temporal notation [25]. These are: **trash_ltl**, temporal specification of a file system trash can; and **trains_ltl**, a temporal model of a train station. The more complex models from the literature, which we refer to as **Complex**, are: **https**, a specification of web attacks based on credentials stealing (taken from Alloy's website); **dijkstra**, a

TABLE I
FEATURES OF THE ANALYZED CASE STUDIES

Benchmark	Model	LOC	sig/rel	fact	pred/fun	faulty
A4F Static	class_fol	91	5/3	0	15/0	2070
	class_rl	87	5/3	0	15/0	1881
	courses	98	5/5	0	15/0	6719
	cv	44	5/4	0	4/0	486
	cv2	44	5/4	0	4/0	176
	graph	70	1/1	0	8/0	644
	lts	61	3/1	0	7/0	901
	network	60	5/4	0	8/0	6714
	prod	75	10/4	0	10/0	2758
	prod2	45	5/3	0	4/0	224
	trash_fol	88	3/1	0	10/0	520
	trash_rl	82	3/1	0	10/0	719
	trains	63	7/2	0	10/0	3252
Complex	chord	503	12/9	22	29/0	-
	dijkstra	84	3/2	0	8/0	-
	https	133	11/12	3	4/1	-
A4F Temporal	trains_ltl	119	7/3	1	18	667
	trash_ltl	117	3/1	0	20	2215

model of Dijkstra's mutual exclusion algorithm (taken from Alloy Analyzer 4); and **chord**, Pamela Zave's specification of Chord's ring maintenance protocol [47]. Table I summarizes the key features of the specifications: lines of code (LOC), number of signatures (sig), relations (rel), facts (fact), predicates (pred), functions (fun), and, for Alloy4Fun subjects, the corresponding number of faulty versions (faulty). Alloy4Fun faulty specifications are student submissions, each possibly containing multiple faults, in different predicates. To isolate the effect of errors in different predicates, from each faulty specification, we generated a version of the specification for each fault where only a single predicate was faulty, and all others were corrected (we also removed syntactic duplicates).

We compare COMBA_1, COMBA_2 and COMBA_3 (C_1, C_2, C_3 for short) with related test generation approaches for Alloy, namely AUnit (`AU`), MuAlloy (`MuA`), and the partitioning scheme introduced in [37], which we refer to as "Scenario Tour" (`ST`). While, as opposed to AUnit and MuAlloy, `ST` is not technically presented as a test generation technique for Alloy, its instance enumeration approach draws ideas from combinatorial testing, as in our approach, and thus deserves a comparison and a more detailed description. Basically, `ST` considers, for each relational field `r: A -> B` in a model, different cardinality constraints on `r`. These constraints enforce constructing different instances where there exists an element of `A`: *(i)* not related to any value through `r` (called *None*), *(ii)* related to exactly one value through `r` (called *One*), and *(iii)* related to more than one value through `r` (called *MTE2*). Similar constraints are enforced on elements of `B` related to elements of `A` by the transpose of `r`, leading to a total of six cardinality constraints per relational field [37]. The constraints on individual fields are combined to form test requirements. While [37] refers to the combination as "pairwise", the actual test requirements correspond in fact to "all combinations" (see Fig. 3, page 47 of [37]). Since we could not find a replication package for `ST`, we simulated the approach (excluding negative instance generation, which as described in [37] requires manual intervention) in our COMBA prototype, by creating and combining the `ST` test requirements described above. There is no obvious adaptation of `ST` for temporal models, where the `Time` signature is involved in mutable fields, and thus most of the cardinality constraints become unfeasible. We thus exclude `ST` of the analysis of temporal models.

Since both our approach and `ST` are inspired by combinatorial testing, it is important to stress the similarities and differences. `ST` first introduced cardinality based partitioning, but the test classes are defined solely for fields, with six classes per field. Our approach uses a coarser grained partitioning (only two classes), but these are not restricted to fields; instead, these are defined for various specification elements, including signatures, fields, predicates and functions. Also, `ST` combines these classes in an "all combinations" fashion, whereas our approach offers the traditional flexibility of combinatorial testing [5]. Our approach also incorporates a fully automated mechanism for generating negative tests, by systematically negating facts and contrasting satisfiability results with respect to the original specification. Finally, while `ST` employs standard SAT solving, our approach exploits a more sophisticated SAT mechanism, based on a novel usage of incremental SAT solving.

To assess the bug-detection effectiveness of test generation approach `TG` in specification `S`, we perform a regression analysis. This is the traditional way to assess test generation approaches for software [18], [19], [38]. We use `TG` to generate test cases in the correct version of `S`, and then run all tests in each faulty variant `S'` of `S`. If at least one test fails, the error in `S'` is revealed by `TG`. In other words, we analyze whether tests generated by `TG` are capable of detecting regressions in `S` (bugs introduced in `S` leading to `S'`).

Since the specifications in **Complex** lack faulty variants, we introduce artificial bugs into the correct specifications via mutations, using MuAlloy (this clearly favors MuAlloy in the experiments). Furthermore, we also employ the DeepSeek large language model [1] to generate faulty versions of these specifications, using prompts that request the introduction of human-like faults in the specifications. We limit the LLM based generation of faulty specifications to 10 specifications per model. The reason is that, in our experiments, the quality of the specifications produced after the 10th request was typically very low, generally producing either syntactically incorrect specifications, specifications with very drastic semantic changes, or simple mutations already captured by MuAlloy, and thus redundant for our experiments.

All experiments were conducted on an Intel Core i7-11700 workstation with 32GB RAM, enforcing a one-hour time limit per execution. We evaluated scopes from 3 to 6, as `MuA` cannot generate tests within the time constraint for larger scopes. Notice that temporal model comparisons exclude `AU` due to its lack of support for temporal specifications. Extended experimental results and a replication package for the experiments are available online [2].

TABLE II
AVERAGE TEST GENERATION TIMES FOR VARIOUS SCOPES (IN SECONDS)

Specs	Scope	MuA	AU	C_1	C_2	C_3	ST
A4F Static	3	1.6	1.3	0.3	0.3	0.4	6
	4	1.8	1.3	0.3	0.3	0.4	8.2
	5	2.9	1.3	0.3	0.4	0.5	9.4
	6	270.6	1.4	0.3	0.4	0.6	10.9
Complex	3	27.8	136.9	0.8	1.3	4	1336
	4	41.7	220.8	1.2	8.9	45.4	1441.3
	5	89.3	265.3	1.8	32.1	1205.5	2400.2
	6	377.2	410.8	3.4	30.5	2400.3	2400.2
A4F Temporal	3	464.9	-	0.9	1.2	1.9	-
	4	1825.9	-	1.6	1.9	2.7	-
	5	1830.8	-	5.5	3.3	4.9	-
	6	1835.3	-	3.7	5.5	12.3	-

TABLE III
AVERAGE TEST SUITE SIZES FOR SEVERAL SCOPES

Specs	Scope	MuA	AU	C_1	C_2	C_3	ST
A4F Static	3	78	55.1	8.6	20.2	51.2	28.3
	4	76.1	54.4	8.1	21	55	16.3
	5	78.3	55.3	8.1	21.5	57.6	17.3
	6	80.8	54.6	8.3	21.8	60.5	20.4
Complex	3	231.6	79.3	24	61.6	203.6	TO
	4	257	75	28.3	68	224.6	TO
	5	278.3	82.3	25.6	67	TO	TO
	6	286.3	88	28	63.6	TO	TO
A4F Temporal	3	611.5	-	19	42	123	-
	4	TO	-	18.5	44.5	113	-
	5	TO	-	18.5	46	125.5	-
	6	TO	-	17.5	43	116	-

B. Results

a) RQ1: Efficiency: Table II summarizes the average test generation times in seconds for the assessed approaches and increasingly large scopes, on the evaluated subjects. Unsurprisingly, `MuA`, `ST` and C_3 show the worse runtime performance, even for the smaller Alloy4Fun static models. Also, these techniques noticeably suffer from scalability, as scopes are increased, especially for the **Complex** and Alloy4Fun temporal models (when applicable). `Au` has a reasonable performance for the Alloy4Fun static models (recall that it is inapplicable to the temporal models), but starts to show some serious performance limitations for the **Complex** models, being even worse than `MuA`. It is worth mentioning that the average times reported for the less scalable techniques, `MuA`, `ST`, and for some scopes C_3, include timeout cases (for those cases, we computed the timeout time as the analysis time, while the actual analysis time may be larger; timeout cases are further reported in the following subsection on test suite size). Overall, C_1 is in general the fastest of the assessed approaches, and the one with less scalability issues as scopes grow. C_2 is also fast in comparison with the other techniques. In particular, for Alloy4Fun models, both C_1 and C_2 are able to generate test suites in less than 6 seconds (on average), for all the analyzed scopes, significantly below the other techniques. Even for the **Complex** models, C_2 shows good performance, generating test suites in about 30 seconds, on average (the closest competing technique is `Au`, taking on average over 6.5 minutes).

TABLE IV
BUG DETECTION (%) IN A4F FAULTY SPECIFICATIONS

Specs	Scope	MuA	AU	C_1	C_2	C_3	ST
A4F Static	3	87.6	77.1	83.3	93.9	96.2	78.9
	4	85.3	78.6	83.5	94.2	96.7	65.1
	5	86.7	78.1	81.5	94.6	96.9	64.8
	6	86.5	78.5	81.9	93.8	97.5	61.1
A4F Temporal	3	94.5	-	91.7	98.9	99.6	-
	4	43.9	-	93.1	99.2	99.8	-
	5	41.5	-	89	99.2	99.8	-
	6	40.9	-	91.5	98.9	99.8	-

b) RQ2: Number of tests: Table III shows the average number of tests generated by each approach, for scopes ranging from 3 to 6. As the Table illustrates, for each technique, its corresponding test suite sizes do not change as scopes are increased, since the test requirements of each of the approaches is independent of the scope for analysis. Across techniques we observe that both `MuA` and `Au` lead to the largest test suites on average, with C_3 also showing large (although slightly smaller than the other two) suites. `ST` has test suite sizes that are comparable to C_2, for **A4F Static**.

Given the above observation, and since some approaches exceed the timeout for larger scopes (`MuA` in **A4F Temporal**, C_3 in **Complex**), let us focus our comparison on test suite size solely for scope 3, where the majority of the techniques are able to fully generate their corresponding test suites without timeouts. C_1 produces test suites that are at leat 69% smaller than those of `AU` (**Complex**), while C_2 generates suites that are at least 22% smaller than `AU`'s. In relation to `MuA`, C_1 generates suites at least 88% smaller (**A4F Static**), and C_2 produces suites at least 73% smaller (**Complex**). With respect to `ST`, C_1 generates suites 69% smaller, and C_2 produces suites 28% smaller (in **A4F Static** only, since `ST` times out for scope 3 in **Complex**). C_3 generates significantly more tests than C_2. Still, C_3's test suites are smaller than `MuA`'s suites in all cases. With respect to `AU`, C_3's test suites are 153% larger in **Complex**. In **A4F Static**, C_3's test suites are 80% larger than those of `ST`.

To put in perspective the test suite size in relation to the complexity of the specifications, we may also analyze the average number of tests per specification (avg. size in Table III) divided by the number of specification components (predicates, functions and facts). In the worst case (**Complex**), C_1 generates about 1.3 tests per component, and C_2 generates 2.8. C_3's test suites are significantly larger, consisting of 7.1 tests per component in the worst case. `AU` lies in between C_2 and C_3, 5.8 tests per component in the worst case (**A4F Static**); recall that `AU` is inapplicable to temporal models. Finally, `MuA`'s cost per component is significantly larger: 31.6 tests by component in the worst case, `A4F Temporal`.

c) RQ3: Bug finding performance: Table IV presents the fault detection percentage achieved on average by the assessed approaches. C_2 and C_3 exhibit a clearly superior bug-finding performance, they find over 94% and 96% of the 29.946 bugs

TABLE V
BUG DETECTION IN LLM-GENERATED FAULTS AND MUTATION SCORE FOR **COMPLEX**

Bug Detection in LLM-Generated Faults							
Model	MuA	AU	C_1	C_2	C_3	ST	# bugs
chord	3	7	9	9	TO	TO	10
dijkstra	10	9	9	10	TO	6	10
https	8	8	8	8	8	TO	10
average (%)	70	80	86.66	90	-	-	-
Mutation Score							
chord	100	99.74	99.74	99.74	TO	TO	797
dijkstra	100	76.87	85.71	92.51	TO	60.54	147
https	100	100	100	100	100	TO	97
average (%)	100	92.2	95.15	97.41	-	-	-

in this benchmark. The next best tool is `MuA`, followed by C_1. `AU` and `ST` show the worst performance here.

Taking into account generation times, size of the test suites, and bug-finding performance, C_2 is overall the best approach. It finds 7.1 and 4.7% more bugs than the best existing approach `MuA`, with suites 74% and 93% smaller, in **A4F Static** and **A4F Temporal**, respectively. It also finds 21,7% more bugs than `AU` in **A4F Static**, with 63% less tests. C_2 is also significantly faster and scales better w.r.t. the complexity of the specifications and the scopes, compared to `MuA` and `AU`.

C_1 is a good intermediate approach, since it achieves good bug-finding results (it finds above 83% of the bugs) with very small test suites and very fast run times (less than 4 seconds on average in all cases). It finds 5% and 3% less bugs than `MuA`, but its suites are substantially smaller: 88% and 93% smaller, in **A4F Static** and **A4F Temporal**, respectively. In addition, C_1 clearly outperforms `AU`, finding 8% more bugs with 69% less tests in **A4F Static**. C_1 also generates much smaller suites than C_2: 57 and 61% smaller in **A4F Static** and **A4F Temporal**, respectively.

C_3 finds a few more bugs than C_2 but at the cost of generating 153 to 192% larger suites **A4F Static** and **A4F Temporal**, respectively, and suffering from scalability problems in larger models (see below). We believe the disadvantages of C_3 w.r.t. C_2 greatly outweigh its benefits.

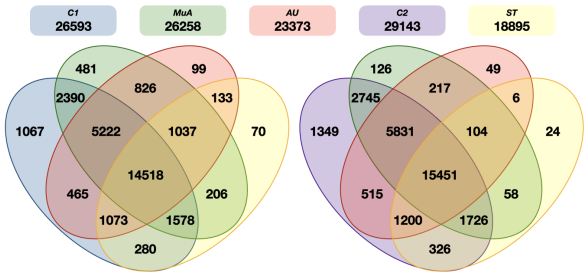

Fig. 7. Comparison of overlapping and unique faults detected for C_1 (left) and C_2 (right), in relation to other techniques

Figure 7 graphically illustrates the overlapping and unique faults detected by C_1 (left) and C_2 (right), in relation to other techniques. Due to a lack of space, we only discuss C_2 briefly here. As expected from the bug finding results, C_2 finds more unique bugs than the other approaches (1349), followed by `MuA` (126), `AU` (49) and finally `ST` (24). C_2 uniquely finds bugs requiring interaction of the components of the specification. In particular, a combination of predicates is found to be effective for finding errors. We mention here a few examples of these situations, corresponding to bugs uniquely found by C_2. The illustrative example (Section III) shows a bug that is discovered when combining a cardinality partition (`no Institution`) with a predicate (`owner`). In `graph`, predicate `faulty_weaklyConnected` incorrectly requires strong connectivity, and C_2 combines its negation with the `oriented` predicate to generate a witness for the error. In `trains_ltl`, the intended property is underspecified, and `faulty_prop14` omits "or leaves station" from the conditions that should make the signal change from green to a different color; combined with `prop2` (all signals are eventually green), C_2 generates tests where a train leaves the station and the signal is kept green, revealing the error.

Table V presents the but finding results for complex specifications (recall that these are based on mutations as artificial bugs). `MuA` achieves perfect mutation score for mutants generated by MuAlloy, as expected. Both C_1 and C_2 perform very well, surpassing **AU** and `ST`, and achieving a score of 95% and 97%, respectively. In addition, Table V shows results for complex models with LLM-generated bugs. Here, C_1 and C_2 outperform all other approaches, in every case. C_2 maintains the best balance overall between efficiency, size of the test suites, and bug-finding performance.

d) RQ4: Impact of incremental SAT solving: For this RQ we compared two COMBA variants: one employing incremental solving and another using a traditional SAT approach (that has to produce a new CNF formula for each test requirement). The results show that the incremental versions of C_1, C_2 and C_3 run 6.5x, 11.1x and 38.3x faster (for the larger assessed scopes where the approaches do not timeout, scope 6 for C_1 and C_2, and scope 4 for C_3), respectively, in comparison to the versions that employ traditional SAT solving. Clearly, incremental SAT solving is crucial for the efficiency of COMBA.

e) RQ5: Unfeasible test requirements: We report the average of unfeasible test requirements generated by COMBA for **Complex**, which is the worst case because the models have many facts (only a few of the A4F cases have facts). On average, C_1, C_2 and C_3 generate 12% (7.6 avg.), 33% (329 avg.) and 64% (19443 avg.) unfeasible test requirements (again, for scopes 6 for C_1 and C_2, 4 for C_3). The very large number of unfeasible requirements created by C_3 explains its bad performance in the complex specifications.

VI. RELATED WORK

Specification instances have been used as input for model validation [48], to analyze software [3], [10], [31], [20], and to implement executable contracts [30], among other applications. For example, there are many test generation approaches

for software that employ Alloy specifications to generate *test inputs* [31], [3], [20]. Alloy instances have also been employed for automated repair and synthesis of Alloy models [44], [7], [16], as well as for fault localization [21], [49].

Other works deal with the selection of instances to be presented to Alloy users. HawkEye is designed to assist the user in querying for new instances with particular characteristics [39]. Aluminum's approach consists of presenting the user with minimal instances [29]. Other works propose abstract instances [35]: abstract representations of families of concrete instances that summarize common aspects of related instances. These approaches aid the user in assessing the correctness of a specification by reducing the number of reported instances, often by focusing in instances with specific characteristics.

COMBA differs from manual inspection of instances in several crucial aspects. Inspection of instances is often subjective: there is no clear termination criteria, and provides no guidance on what component of the specification to test next. Hence, the user might stop looking at instances before finding an error in some component. In contrast, systematic testing approaches like COMBA (also AUnit, MuAlloy) provide a clear termination criteria for testing, and its thorough coverage of the specification results in good bug-finding performance (as evidenced by our experiments). COMBA tests all the individual components of the specification, and interactions between the components (when tuples regarding different components are covered). This is why COMBA does not take into account run/check commands for the generation. The most closely related systematic testing approaches, AUnit [40] and `MuAlloy` [41], [43], [17], are comprehensively assessed against COMBA in Sec. V.

Another related approach is CompoSAT [34], which introduces a coverage criterion for Alloy, guided by the specification. However, the focus is not on test generation, but rather on provenances: essentially, explanations of the necessary constituent elements in satisfying instances. In addition, CompoSAT only deals with positive test scenarios, as opposed to our approach. COMBA's goal is different, as it aims at generating test scenarios for Alloy specifications, involving both positive and negative cases, and achieving a thorough examination of the specification. In addition, COMBA leverages on advanced SAT-solving techniques.

The work in [31], [33] delves into the domain of field-exhaustive input generation for specification-based black-box testing for software. These approaches exploit the incremental SAT solving approach of adding clauses iteratively, in order to speed up the generation of test inputs. As mentioned earlier, adding clauses incrementally would not work for COMBA, as once a clause is added it cannot be removed without restarting the solver. Besides being applied to a different problem, COMBA presents an innovative use of the mechanisms of modern incremental SAT solvers, namely employing "solve with assumptions" [9], [28] to achieve the desired results.

Many approaches in the literature deal with the problem of generating "minimal" test suites that achieve combinatorial coverage [24], [23], [22], [46], [42]. Some of the approaches employ heuristics for fast sampling of a set of instances that cover many of the test requirements [24], some approaches use greedy algorithms [46], other approaches employ meta-heuristics [23], [22], etc. The approach of [42] addresses the problem of keeping the combinatorial test suites of evolving software up to date, and strives to maintain small test suites. While the generation of minimal test suites is very important, our experimental evaluation shows that the approaches C_1 and C_2 already produce small enough test suites. For future work, we plan to incorporate some of the approaches for minimizing test suites into COMBA, and make COMBA scale better to larger combinations of parameters (e.g., $t > 2$). For example, we could incorporate into COMBA the sampling approach of [24], or the elimination of unfeasible tuples based on UNSAT cores presented in [46].

VII. Conclusions

We introduced COMBA, a novel combinatorial test generation approach for Alloy specifications. A key characteristic of COMBA is that it partitions the state space of an Alloy specification in an automated way (without user assistance). In this way, COMBA defines a systematic way to produce a relatively small set of test cases to assess the correctness of the specification. Remarkably, COMBA_2 generates smaller test suites than existing test generation approaches for Alloy, that find a significantly larger number of bugs. In addition, thanks to its efficient test generation algorithm (which heavily exploits incremental SAT solving), COMBA_2 generates tests faster than existing approaches.

Being a combinatorial approach, an advantage of COMBA over existing approaches is that it can be configured to produce test suites of different sizes. Stronger COMBA criteria (using a larger t) give increasing levels of thoroughness in the coverage of the interactions between the specification components, thus improving the confidence on the correctness of the specification, but at the cost of producing a larger number of tests.

Note that, unlike combinatorial approaches for software testing where often the different partitions are created for different components (e.g., different partitions for different methods), COMBA generates a unique partitioning scheme for the whole specification. In other words, COMBA aims to generate test cases for *all* the components (signatures, relations, predicates, facts, asserts) of the specification. It does so by creating test requirements that involve partitions from distinct components, therefore testing the combination of structural properties of the specification (e.g., based on the cardinalities of signatures/relations) with semantic properties (e.g., using the predicates/assertions).

We believe COMBA could be a good fit for automated specification repair and synthesis techniques that require tests to work [44], [7], [16]. The relatively low number of tests generated, and capacity of varying the size of the generated test suites could be key in this context. We will explore this topic in future work.

REFERENCES

[1] Deepseek's website. https://www.deepseek.com/. Accessed: 2025.

[2] Replication package for *Automated Combinatorial Testing for Alloy*. https://sites.google.com/view/combinatorial-testing-alloy/. Accessed: 2025.

[3] Pablo Abad, Nazareno Aguirre, Valeria S. Bengolea, Daniel Ciolek, Marcelo F. Frias, Juan P. Galeotti, Tom Maibaum, Mariano M. Moscato, Nicolás Rosner, and Ignacio Vissani. Improving test generation under rich contracts by tight bounds and incremental SAT solving. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*, pages 21–30. IEEE Computer Society, 2013.

[4] Jean-Raymond Abrial. *The B-book - assigning programs to meanings*. Cambridge University Press, 1996.

[5] Paul Ammann and Jeff Offutt. *Introduction to Software Testing, 2nd ed.* Cambridge University Press, 2016.

[6] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[7] Simón Gutiérrez Brida, Germán Regis, Guolong Zheng, Hamid Bagheri, ThanhVu Nguyen, Nazareno Aguirre, and Marcelo F. Frias. ICEBAR: feedback-driven iterative repair of alloy specifications. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, pages 55:1–55:13. ACM, 2022.

[8] Greg Dennis, Felix Sheng-Ho Chang, and Daniel Jackson. Modular verification of code with SAT. In Lori L. Pollock and Mauro Pezzè, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*, pages 109–120. ACM, 2006.

[9] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[10] Juan P. Galeotti, Nicolás Rosner, Carlos Gustavo López Pombo, and Marcelo F. Frias. TACO: efficient sat-based bounded verification using symmetry breaking and tight bounds. *IEEE Trans. Software Eng.*, 39(9):1283–1307, 2013.

[11] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[12] Daniel Jackson. *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006.

[13] Daniel Jackson. Alloy: a language and tool for exploring software designs. *Commun. ACM*, 62(9):66–76, 2019.

[14] Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. Alcoa: the alloy constraint analyzer. In Carlo Ghezzi, Mehdi Jazayeri, and Alexander L. Wolf, editors, *Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000*, pages 730–733. ACM, 2000.

[15] Clifford B. Jones. *Systematic software development using VDM (2. ed.)*. Prentice Hall International Series in Computer Science. Prentice Hall, 1991.

[16] Ana Jovanovic and Allison Sullivan. Towards automated input generation for sketching alloy models. In *10th IEEE/ACM International Conference on Formal Methods in Software Engineering, FormaliSE-ICSE 2022, Pittsburgh, PA, USA, May 22-23, 2022*, pages 58–68. ACM, 2022.

[17] Ana Jovanovic and Allison Sullivan. Mutation testing for temporal alloy models. In *26th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2023, Västerås, Sweden, October 1-6, 2023*, pages 228–238. IEEE, 2023.

[18] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In Corina S. Pasareanu and Darko Marinov, editors, *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 437–440. ACM, 2014.

[19] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey, editors, *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 654–665. ACM, 2014.

[20] Shadi Abdul Khalek, Guowei Yang, Lingming Zhang, Darko Marinov, and Sarfraz Khurshid. Testera: A tool for testing java programs using alloy specifications. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 608–611, Washington, DC, USA, 2011. IEEE Computer Society.

[21] Tanvir Ahmed Khan, Allison Sullivan, and Kaiyuan Wang. Alloyfl: a fault localization framework for alloy. In Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta, editors, *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, pages 1535–1539. ACM, 2021.

[22] Jinkun Lin, Shaowei Cai, Bing He, Yingjie Fu, Chuan Luo, and Qingwei Lin. Fastca: An effective and efficient tool for combinatorial covering array generation. In *43rd IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2021, Madrid, Spain, May 25-28, 2021*, pages 77–80. IEEE, 2021.

[23] Jinkun Lin, Chuan Luo, Shaowei Cai, Kaile Su, Dan Hao, and Lu Zhang. TCA: an efficient two-mode meta-heuristic algorithm for combinatorial test generation (T). In Myra B. Cohen, Lars Grunske, and Michael Whalen, editors, *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 494–505. IEEE Computer Society, 2015.

[24] Chuan Luo, Qiyuan Zhao, Shaowei Cai, Hongyu Zhang, and Chunming Hu. Samplingca: effective and efficient sampling-based pairwise testing for highly configurable software systems. In Abhik Roychoudhury, Cristian Cadar, and Miryung Kim, editors, *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pages 1185–1197. ACM, 2022.

[25] Nuno Macedo, Julien Brunel, David Chemouil, Alcino Cunha, and Denis Kuperberg. Lightweight specification and analysis of dynamic systems with rich configurations. In Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su, editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 373–383. ACM, 2016.

[26] Nuno Macedo, Alcino Cunha, and Ana C. R. Paiva. Alloy4fun dataset for 2022/23. https://doi.org/10.5281/zenodo.8123547, Jul 2023.

[27] Nuno Macedo, Alcino Cunha, José Pereira, Renato Carvalho, Ricardo Silva, Ana C. R. Paiva, and Miguel Sozinho Ramalho anbd Daniel Castro Silva. Experiences on teaching alloy with an automated assessment platform. In Alexander Raschke, Dominique Méry, and Frank Houdek, editors, *Rigorous State-Based Methods - 7th International Conference, ABZ 2020, Ulm, Germany, May 27-29, 2020, Proceedings*, volume 12071 of *Lecture Notes in Computer Science*, pages 61–77. Springer, 2020.

[28] Alexander Nadel and Vadim Ryvchin. Efficient SAT solving under assumptions. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 242–255. Springer, 2012.

[29] Tim Nelson, Salman Saghafi, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Aluminum: principled scenario exploration through minimality. In David Notkin, Betty H. C. Cheng, and Klaus Pohl, editors, *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 232–241. IEEE Computer Society, 2013.

[30] Pengyu Nie, Marinela Parovic, Zhiqiang Zang, Sarfraz Khurshid, Aleksandar Milicevic, and Milos Gligoric. Unifying execution of imperative generators and declarative specifications. *Proc. ACM Program. Lang.*, 4(OOPSLA):217:1–217:26, 2020.

[31] Pablo Ponzio, Nazareno Aguirre, Marcelo F. Frias, and Willem Visser. Field-exhaustive testing. In Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su, editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 908–919. ACM, 2016.

[32] Pablo Ponzio, Nazareno Aguirre, Marcelo F. Frias, and Willem Visser. Field-exhaustive testing. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 908–919, New York, NY, USA, 2016. ACM.

[33] Pablo Ponzio, Ariel Godio, Nicolás Rosner, Marcelo Arroyo, Nazareno Aguirre, and Marcelo F. Frias. Efficient bounded model checking of heap-manipulating programs using tight field bounds. In Esther Guerra and Mariëlle Stoelinga, editors, *Fundamental Approaches to Software Engineering - 24th International Conference, FASE 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*, volume 12649 of *Lecture Notes in Computer Science*, pages 218–239. Springer, 2021.

[34] Sorawee Porncharoenwase, Tim Nelson, and Shriram Krishnamurthi. Composat: Specification-guided coverage for model finding. In Klaus Havelund, Jan Peleska, Bill Roscoe, and Erik P. de Vink, editors, *Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings*, volume 10951 of *Lecture Notes in Computer Science*, pages 568–587. Springer, 2018.

[35] Jan Oliver Ringert and Allison Sullivan. Abstract alloy instances. In Marsha Chechik, Joost-Pieter Katoen, and Martin Leucker, editors, *Formal Methods - 25th International Symposium, FM 2023, Lübeck, Germany, March 6-10, 2023, Proceedings*, volume 14000 of *Lecture Notes in Computer Science*, pages 364–382. Springer, 2023.

[36] Nicolás Rosner, Jaco Geldenhuys, Nazareno Aguirre, Willem Visser, and Marcelo F. Frias. BLISS: improved symbolic execution by bounded lazy initialization with SAT support. *IEEE Trans. Software Eng.*, 41(7):639–660, 2015.

[37] Takaya Saeki, Fuyuki Ishikawa, and Shinichi Honiden. Automatic generation of potentially pathological instances for validating alloy models. In Kazuhiro Ogata, Mark Lawford, and Shaoying Liu, editors, *Formal Methods and Software Engineering - 18th International Conference on Formal Engineering Methods, ICFEM 2016, Tokyo, Japan, November 14-18, 2016, Proceedings*, volume 10009 of *Lecture Notes in Computer Science*, pages 41–56, 2016.

[38] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (T). In Myra B. Cohen, Lars Grunske, and Michael Whalen, editors, *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 201–211. IEEE Computer Society, 2015.

[39] Allison Sullivan. Hawkeye: User-guided enumeration of scenarios. In Zhi Jin, Xuandong Li, Jianwen Xiang, Leonardo Mariani, Ting Liu, Xiao Yu, and Nahgmeh Ivaki, editors, *32nd IEEE International Symposium on Software Reliability Engineering, ISSRE 2021, Wuhan, China, October 25-28, 2021*, pages 569–578. IEEE, 2021.

[40] Allison Sullivan, Kaiyuan Wang, and Sarfraz Khurshid. Aunit: A test automation tool for alloy. In *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*, pages 398–403. IEEE Computer Society, 2018.

[41] Allison Sullivan, Kaiyuan Wang, Razieh Nokhbeh Zaeem, and Sarfraz Khurshid. Automated test generation and mutation testing for alloy. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*, pages 264–275. IEEE Computer Society, 2017.

[42] Rachel Tzoref-Brill and Shahar Maoz. Modify, enhance, select: coevolution of combinatorial models and test plans. In Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu, editors, *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 235–245. ACM, 2018.

[43] Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. Mualloy: a mutation testing framework for alloy. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 29–32. ACM, 2018.

[44] Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. Arepair: a repair framework for alloy. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 103–106. IEEE / ACM, 2019.

[45] J. C. P. Woodcock and Jim Davies. *Using Z - specification, refinement, and proof*. Prentice Hall international series in computer science. Prentice Hall, 1996.

[46] Akihisa Yamada, Armin Biere, Cyrille Artho, Takashi Kitamura, and Eun-Hye Choi. Greedy combinatorial test case generation using unsatisfiable cores. In David Lo, Sven Apel, and Sarfraz Khurshid, editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 614–624. ACM, 2016.

[47] Pamela Zave. Pamela zave's chord algorithm specification.

[48] Pamela Zave. Reasoning about identifier spaces: How to make chord correct. *IEEE Trans. Software Eng.*, 43(12):1144–1156, 2017.

[49] Guolong Zheng, ThanhVu Nguyen, Simón Gutiérrez Brida, Germán Regis, Marcelo F. Frias, Nazareno Aguirre, and Hamid Bagheri. FLACK: counterexample-guided fault localization for alloy models. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 637–648. IEEE, 2021.