Validating Formal Specifications with LLM-generated Test Cases

Alcino Cunha & Nuno Macedo
INESC TEC & University of Minho, Portugal

Motivation

```
sig Person {
  teaches : set Course,
  enrolled : set Course
}
sig Professor in Person {}
sig Student in Person {}
sig Course {}
```

```
fact Assumptions {
  // There are no people besides professors and students
  Person = Professor + Student
  // All courses have professors
  all c : Course | some teaches.c
  // Only professors teach
  teaches.Course in Professor
  // No one teaches themselves
  all p : Person | no p.teaches & p.enrolled
}
```

Only students can be enrolled

```
fact Student1 {
  enrolled.Course in Student
}
fact Student2 {
  all p: Person | some p.enrolled implies p in Student
}
fact Student3 {
  no Professor.enrolled
}
fact Student4 {
  enrolled in Student -> Course
}
```

/Users/alcino/

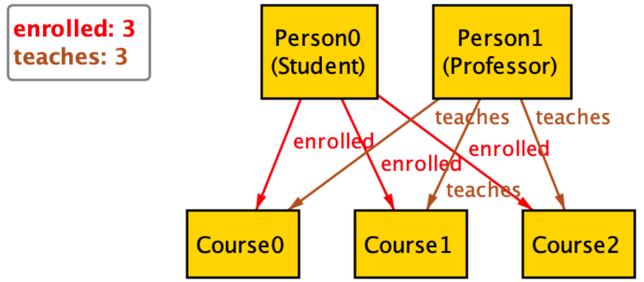
New Open Reload Save Execute Show

```
  teaches : set Course,
  enrolled : set Course
}
sig Professor in Person {}
sig Student in Person {}
sig Course {}

fact Assumptions {
    // There are no people besides professors and stude
    Person = Professor + Student
    // All courses have professors
    all c : Course | some teaches.c
    // Only professors teach
    teaches.Course in Professor
    // No one teaches themselves
    all p : Person | no p.teaches & p.enrolled
}

fact OnlyStudentsEnrolled {
  no Professor.enrolled
}

run {}
```

Line 45, Column 1 [modified]

(Talk) Run run$1

Viz Txt Table Tree Theme Magic Layout Evaluator New Projection: none




```
Executing "Run run$1"
   Actual scopes: 3 Person, 3 Course
   Solver=minisat Bitwidth=4 MaxSeq=4 SkolemDepth=1 S
   306 vars. 30 primary vars. 468 clauses. 3ms.
   Instance found. Predicate is consistent. 3ms.
```

/Users/alcino/

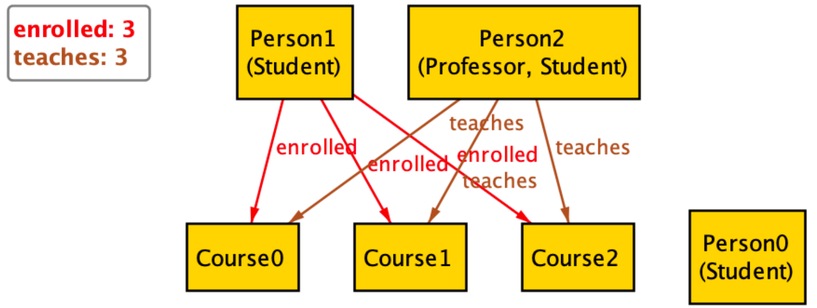
New Open Reload Save Execute Show

```
  teaches : set Course,
  enrolled : set Course
}
sig Professor in Person {}
sig Student in Person {}
sig Course {}

fact Assumptions {
    // There are no people besides professors and stude
    Person = Professor + Student
    // All courses have professors
    all c : Course | some teaches.c
    // Only professors teach
    teaches.Course in Professor
    // No one teaches themselves
    all p : Person | no p.teaches & p.enrolled
}

fact OnlyStudentsEnrolled {
  no Professor.enrolled
}

run {}
```

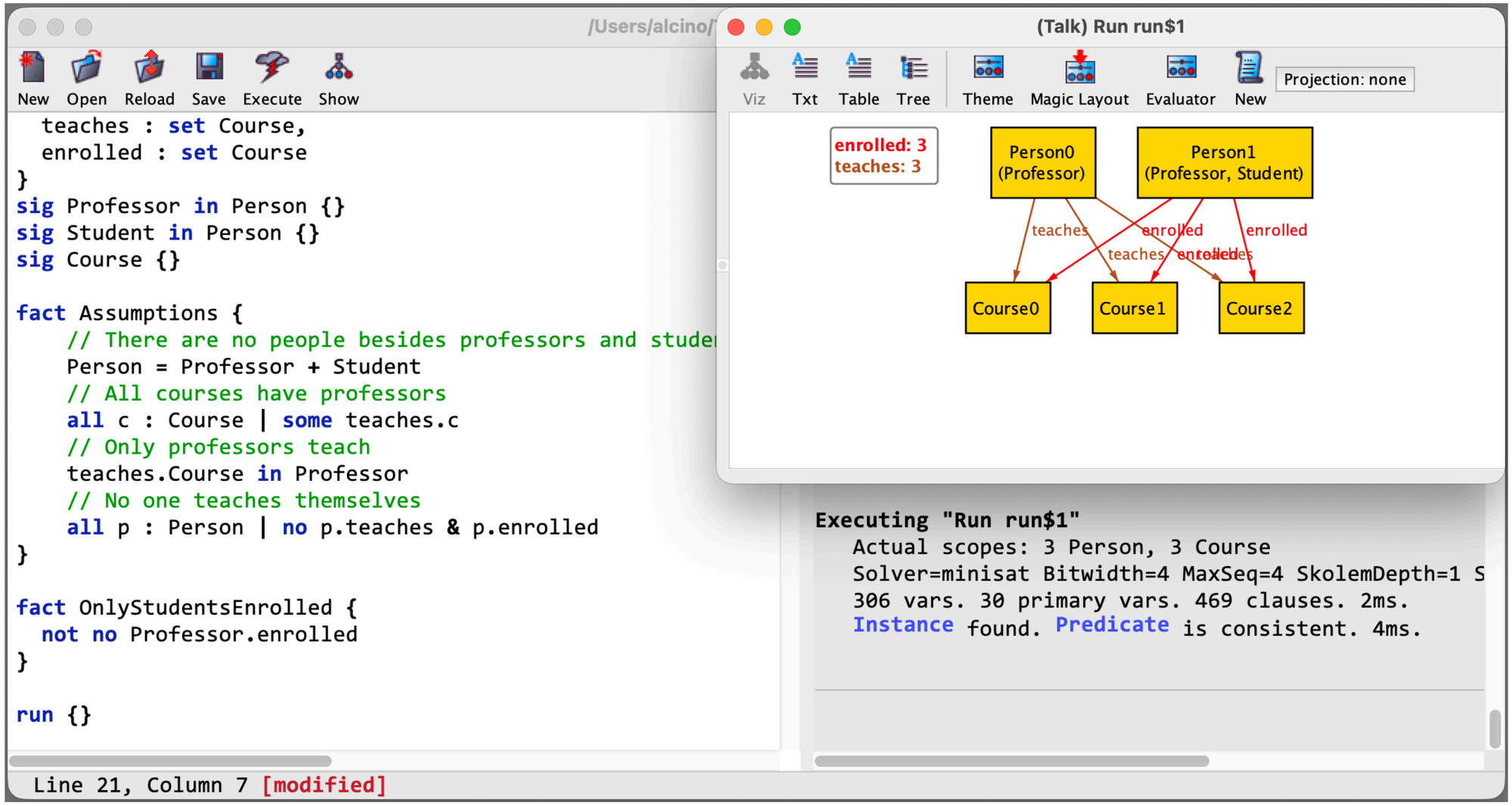
Line 45, Column 1 [modified]

(Talk) Run run$1

Viz Txt Table Tree Theme Magic Layout Evaluator New Projection: none




```
Executing "Run run$1"
   Actual scopes: 3 Person, 3 Course
   Solver=minisat Bitwidth=4 MaxSeq=4 SkolemDepth=1 S
   306 vars. 30 primary vars. 468 clauses. 3ms.
   Instance found. Predicate is consistent. 3ms.
```


/Users/alcino/
New
Open
Reload
Save
Execute
Show
teaches : set Course,
enrolled : set Course
}
sig Professor in Person {}
sig Student in Person {}
sig Course {}
fact Assumptions {
// There are no people besides professors and stude
Person = Professor + Student
// All courses have professors
all c : Course | some teaches.c
// Only professors teach
teaches.Course in Professor
// No one teaches themselves
all p : Person | no p.teaches & p.enrolled
}
fact OnlyStudentsEnrolled {
not no Professor.enrolled
}
run {}
Line 21, Column 7 [modified]
(Talk) Run run$1
Viz
Txt
Table
Tree
Theme
Magic Layout
Evaluator
New
Projection: none
enrolled: 3
teaches: 3
Person0
(Professor)
Person1
(Professor, Student)
teaches
enrolled
enrolled
teaches
enrolled
Course0
Course1
Course2
Executing "Run run$1"
Actual scopes: 3 Person, 3 Course
Solver=minisat Bitwidth=4 MaxSeq=4 SkolemDepth=1 S
306 vars. 30 primary vars. 469 clauses. 2ms.
Instance found. Predicate is consistent. 4ms.

```
// A professor is enrolled in multiple courses
run Negative {
  some disj P1,P2 : Person, disj C1,C2 : Course {
    Person = P1 + P2
    Professor = P1 + P2
    Student = none
    Course = C1 + C2
    teaches = P1->C1 + P1->C2
    enrolled = P2->C1 + P2->C2
  }
} for 2 Person, 2 Course expect 0
```

```
// Only enrolled person is both student and professor
run Positive {
  some disj P1,P2 : Person, disj C1 : Course {
    Person = P1 + P2
    Professor = P1 + P2
    Student = P2
    Course = C1
    teaches = P1->C1
    enrolled = P2->C1
  }
} for 2 Person, 1 Course expect 1
```

```
Executing "Run Negative for 2 Person, 2 Course expect 0"
   Actual scopes: 2 Person, 2 Course
   Solver=minisat Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20 Mode=batch
   289 vars. 24 primary vars. 466 clauses. 2ms.
   No instance found. Predicate may be inconsistent, as expected. 1ms.

Executing "Run Positive for 2 Person, 1 Course expect 1"
   Actual scopes: 2 Person, 1 Course
   Solver=minisat Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=OFF Mode=batch
   429 vars. 40 primary vars. 670 clauses. 1ms.
   No instance found. Predicate may be inconsistent, contrary to expectation. 1ms.

2 commands were executed. The results are:
   #1: No instance found. Negative may be inconsistent, as expected.
   #2: No instance found. Positive may be inconsistent, contrary to expectation.
```

“Can LLMs help us with validation?”

Empirical study

Research questions

1. How does prompt design influence the effectiveness of test case generation?
2. What is the effect of non-determinism in test case generation?
3. How do different LLMs compare in test case generation?
4. What are the characteristics of invalid tests cases?
5. How good are the generated test suites at finding incorrect specifications?

Alloy4Fun

```
1  open util/ordering[Grade]
2
3  sig Person {
4      teaches : set Course,
5      enrolled : set Course,
6      projects : set Project
7  }
8
9  sig Professor,Student in Person {}
10
11 sig Course {
12     projects : set Project,
13     grades : Person -> Grade
14 }
15
16 sig Project {}
17
18 sig Grade {}
19 // Specify the following properties.
20 // You can check their correctness with the different commands and
21 // when specifying each property you can assume all the previous ones to be true.
22
23 pred inv1 {
24     // Only students can be enrolled in courses
25
26 }
27
28
29 pred inv2 {
30     // Only professors can teach courses
31
32 }
33
34
35 pred inv3 {
```

Command : check inv1

Execute

Share model

Statistics

Derivations

To get started with Alloy4Fun check out this introduction, along with some example challenges. More information about Alloy is available at AlloyTools.

Privacy Policy: Alloy4Fun logs all user interactions, including code and options, which may be analyzed and made public for research purposes. Do not share confidential information. However, Alloy4Fun does not collect any personally identifiable information.

Alloy4Fun developed by:

INESCTEC

Universidade do

U.PORTO FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Funded by:

FCT Fundação para a Ciência e a Tecnologia

COMPETE 2020

PORTUGAL 2020

UNIÃO EUROPEIA Fundos Europeus Estruturais e de Investimento

Published October 2025 | Version MFES 24/25

Dataset Open

Alloy4Fun Dataset for 2024/25

Nuno Macedo[1]; Alcino Cunha[2]; Ana C. R. Paiva[1]

Show affiliations

This dataset contains models submitted by students in the Alloy4Fun platform to solve the challenge models from various editions of formal methods courses in the University of Minho (UM) and the University of Porto (UP) between the fall of 2019 and the spring of 2025, totalling around 185.000 entries. Participants include those enrolled in the optional MSc course "Specification and Modelling" (EM) and the mandatory MSc course "Formal Methods in Software Engineering" (MFES) in UM, and the optional MSc course "Formal Methods for Critical Systems" (MFS) in UP. Note that since the challenges' permalinks are publicly available, the dataset may contain submissions from other participants outside the classroom context.

The analysis of the 2021 dataset is reported in the Science of Computer Programming paper "Experiences on Teaching Alloy with an Automated Assessment Platform" (extending the ABZ'20 conference version analysing the 2020 dataset).

Name	Permalink	Courses (Students)	Entries
Trash FOL	sDLK7uBCbgZon3znd	EM 19/20 (~20) & 20/21 (~20), MFS 21/22 (~10) & 22/23 (~10)	4333
Classroom FOL	YH3ANm7Y5Qe5dSYem	EM 19/20 (~20) & 20/21 (~20), MFS 21/22 (~10) & 22/23 (~10)	5106
Trash RL	PQAJE67kz8w5NWJuM	EM 19/20 (~20) & 20/21 (~20)	4929
Classroom RL	zRAn69AocpkmxXZnW	EM 19/20 (~20) & 20/21 (~20)	6813
Graphs	gAeD3MTGCCv8YNTaK	EM 19/20 (~20) & 20/21 (~20)	3382
LTS	zoEADeCW2b2suJB2k	EM 19/20 (~20) & 20/21 (~20)	3597

		Wrong Specifications		
Example	**Requirements**	**Min**	**Max**	**Mean**
Social network	8	44	848	272
Production line	10	28	585	147
Train station	10	36	203	104
Courses	15	33	312	137

Requirement	Wrong
Only students can be enrolled in courses	161
Only professors can teach courses	33
Courses must have teachers	82
Projects are proposed by one course	87
Only students work on projects and projects must have someone working on them	248
Students only work on projects of courses they are enrolled in	168
Students work on at most one project per course	177
A professor cannot teach herself	79
A professor cannot teach colleagues	312
Only students have grades	80
Students only have grades in courses they are enrolled	87
Students have at most one grade per course	87
A student with the highest mark in a course must have worked on a project on that course	291
A student cannot work with the same student in different projects	51
Students working on the same project in a course cannot have marks differing by more than one	109

Generate N positive and N negative instances for the requirement "R_i" for the following model.
All instances must also satisfy the requirements "R_0", ..., and "R_{i-1}".

Zero-shot

```
Alloy is a formal modelling language where that uses sets (named signatures in Alloy) and relations (sets of tuples, named fields in Alloy) to represent information in a domain.

In Alloy we can express requirements about a model using facts specified in relational logic, which is an extension of first-order logic with relational operators. Given a requirement it is useful to have both positive and negative instances to serve as test cases for validation. An instance of a model is a valuation to all declared sets and relations.

Positive and negative instances can be included as test cases in a model using run commands. A run command includes a specification of the instance using relational logic, a scope that states how many elements of each top-level signature exist, and an expectation that states if the run command should be satisfiable (representing a positive instance) or unsatisfiable (representing a negative instance). To specify an instance with logic we can use an existential quantifiers to capture the elements of each signature of the domain, and then state what are the values of each set and relation using an equality with the name of the set or relation in the left hand side and the value in the right hand side. To express that a set or relation is empty we can specify that it is equal to the empty set (none) or empty relation of appropriate arity (for example, none->none if the relation is binary). Since there can be multiple total orders we must disambiguate which next relation is being defined using the <: operator with the signature on the left. The same applies if more that one relation is declared with the same name. The scope of run command only needs to define the number of elements that exist in top-level signatures, those that are not subsets of any other.

In the following interaction you will act as an Alloy expert. I will ask you to generate positive and negative instances for different requirements for a given model.

Here are some instructions:
- Do not attempt to formalize the requirement with a fact, just output the requested instances as run commands.
- I will clearly state how many positive and negative instances you should output.
- All instances should be truly different (be aware that the names of the elements are actually irrelevant).
- Try to produce minimal instances with few elements.
- All instances must include a comment explaining in natural language why it is positive or negative.
- All positive instances should have an expect 1 and all negative an expect 0.
- All instances must define the scopes for all top-level signatures in the model.
- In every instance you must specify the values for all declared sets and relations with an equality restriction.
- To declare the elements of each signature use a some quantifier with the disj keyword to ensure that all elements are different.
- The value of a relation should take into account its arity. For example, if it is a ternary relation its value must be a set of triples.
- If a set or relation is empty you must explicitly state that it is equal to the empty set (none) or empty relation (of the correct arity).
- If there are two relations with the same name you must disambiguate using the <: operator with the domain signature on the left.
- If a model uses ordering on a signature you must also specify the value of the next binary relation using the <: operator to disambiguate. Signatures with ordering cannot be empty so next is never empty.
- If the value of some sets and relations is irrelevant for a given requirement you can just state that they are empty or assign them random values.
- Generate only Alloy code, do not include any explanations outside the code.
```

One-shot

```
Alloy is a formal modelling language where that uses sets (named signatures in Alloy) and relations (sets of tuples, named fields in Alloy) to represent information in a domain. Consider, for example, the following Alloy model.

open util/ordering[Value]
sig Person {
  likes : set Car,
  owns : Car -> lone Value
}
sig Adult in Person {}
abstract sig Car {}
sig Sedan, SUV extends Car {}
sig Value {}

This model declares a set named Person, a subset of Person named Adult, a set name Car, two extensions (disjoint subsets) of Car named Sedan and SUV, and a set named Value. It also declares a binary relation named likes, that associates each Person with the set of Cars it likes, and ternary relation named owns that associates each Person with the set of Cars it owns and the respective value. When declaring a field we can use other multiplicity keywords instead of set: set means there is an arbitrary number of target elements associated with a source element, lone means there is at most one related element, some means there is at least one related element, and one means there is exactly one related element. When extensions are declared, if the parent signature is declared as abstract then it can only contain elements that are also contained in its extensions. In this model we have two disjoint extensions of abstract set Car, meaning all cars are either sedans or SUVs. A model can use the module ordering to impose a total ordering on a signature. In this example Value is a totally ordered signature. The total ordering is defined in a binary relation called next, that is implicitly declared.

In Alloy we can express requirements about a model using facts specified in relational logic, which is an extension of first-order logic with relational operators. Given a requirement it is useful to have both positive and negative instances to serve as test cases for validation. An instance of a model is a valuation to all declared sets and relations.

Positive and negative instances can be included as test cases in a model using run commands. A run command includes a specification of the instance using relational logic, a scope that states how many elements of each top-level signature exist, and an expectation that states if the run command should be satisfiable (representing a positive instance) or unsatisfiable (representing a negative instance). To specify an instance with logic we can use an existential quantifiers to capture the elements of each signature of the domain, and then state what are the values of each set and relation using an equality with the name of the set or relation in the left hand side and the value in the right hand side.  To express that a set or relation is empty we can specify that it is equal to the empty set (none) or empty relation of appropriate arity (for example, none->none if the relation is binary). Since there can be multiple total orders we must disambiguate which next relation is being defined using the <: operator with the signature on the left. The same applies if more that one relation is declared with the same name. The scope of run command only needs to define the number of elements that exist in top-level signatures, those that are not subsets of any other.

For example, if a requirement is "Every person owns a car" the following command specifies a positive instance where this requirement is satisfied.

run Instance1 {
  some disj Person1,Person2,Person3 : Person | some disj Car1, Car2 : Car | some disj Value1,Value2,Value3 : Value {
    Person = Person1 + Person2 + Person3
  Adult = none
    Car = Car1 + Car2
  Sedan = Car1
  SUV = Car2
  Value = Value1 + Value2 + Value3
  likes = none->none
    owns = Person1->Car1->Value1 + Person2->Car2->Value2 + Person3->Car2->Value2
  Value <: next = Value1->Value2 + Value2->Value3
  }
} for 3 Person, 2 Car, 3 Value expect 1

In the above instance we have three people, named Person1, Person2, and Person3, none of them is an Adult. We have two cars, named Car1 and Car2, the former a Sedan and the latter a SUV. We also have three possible Values, being Value1 the lowest possible and Value3 the highest possible. Person1 owns Car1 with value Value1, and Person2 and Person3 together own Car2 with value Value2. No one likes any car, because relation likes is empty.

In the following interaction you will act as an Alloy expert. I will ask you to generate positive and negative instances for different requirements for a given model.

Here are some instructions:
- Do not attempt to formalize the requirement with a fact, just output the requested instances as run commands.
- I will clearly state how many positive and negative instances you should output.
- All instances should be truly different (be aware that the names of the elements are actually irrelevant).
- Try to produce minimal instances with few elements.
- All instances must include a comment explaining in natural language why it is positive or negative.
- All positive instances should have an expect 1 and all negative an expect 0.
- All instances must define the scopes for all top-level signatures in the model.
- In every instance you must specify the values for all declared sets and relations with an equality restriction.
- To declare the elements of each signature use a some quantifier with the disj keyword to ensure that all elements are different.
- The value of a relation should take into account its arity. For example, if it is a ternary relation its value must be a set of triples.
- If a set or relation is empty you must explicitly state that it is equal to the empty set (none) or empty relation (of the correct arity).
- If there are two relations with the same name you must disambiguate using the <: operator with the domain signature on the left.
- If a model uses ordering on a signature you must also specify the value of the next binary relation using the <: operator to disambiguate. Signatures with ordering cannot be empty so next is never empty.
- If the value of some sets and relations is irrelevant for a given requirement you can just state that they are empty or assign them random values.
- Generate only Alloy code, do not include any explanations outside the code.
```

Few-shot

```
Alloy is a formal modelling language where that uses sets (named signatures in Alloy) and relations (sets of tuples, named fields in Alloy) to represent information in a domain. For example, the Alloy model

sig Person {
  owns : set Car
}
sig Car {}

declares a set named Person, a set name Car, and a binary relation named owns that associates each Person with the set of Cars it owns. When declaring a field we can use other multiplicity keywords instead of set: set means there is an arbitrary number of target elements associated with a source element, lone means there is at most one related element, some means there is at least one related element, and one means there is exactly one related element. If no multiplicity is declared it is by default one.

In Alloy we can express requirements about a model using facts specified in relational logic, which is an extension of first-order logic with relational operators. Given a requirement it is useful to have both positive and negative instances to serve as test cases for validation. An instance of a model is a valuation to all declared sets and relations.

Positive and negative instances can be included as test cases in a model using run commands. A run command includes a specification of the instance using relational logic, a scope that states how many elements of each top-level signature exist, and an expectation that states if the run command should be satisfiable (representing a positive instance) or unsatisfiable (representing a negative instance). To specify an instance with logic we can use an existential quantifiers to capture the elements of each signature of the domain, and then state what are the values of each set and relation using an equality with the name of the set or relation in the left hand side and the value in the right hand side.

For example, if a requirement is "Every person owns a car" the following command specifies a positive instance where this requirement is satisfied.

run Instance1 {
  some disj Person1,Person2,Person3 : Person | some disj Car1, Car2 : Car {
    Person = Person1 + Person2 + Person3
    Car = Car1 + Car2
    owns = Person1->Car1 + Person2->Car2 + Person3->Car2
  }
} for 3 Person, 2 Car expect 1

In this instance we have three people, named Person1, Person2, and Person3, two cars, named Car1 and Car2, Person1 owns Car1, and Person2 and Person3 together own Car2.

On the other hand, the following is a negative instance where this requirement is not satisfied, since Person2 does not own a car.

run Instance2 {
  some disj Person1,Person2 : Person | some disj Car1, Car2 : Car {
    Person = Person1 + Person2
    Car = Car1 + Car2
    owns = Person1->Car1 + Person1->Car2
  }
} for 2 Person, 2 Car expect 0

To express that a set or relation is empty you can specify that it is equal to the empty set (none) or empty relation of appropriate arity (for example, none->none if the relation is binary). For example, the following is another example of a trivial positive instance for the requirement "Every person owns a car", since there are no persons.

run Instance3 {
some disj Car1, Car2 : Car {
Person = none
Car = Car1 + Car2
owns = none->none
}
} for 0 Person, 2 Car expect 1

A model can also declare subsets of a signature with keywords in and extends. The difference is that in declares an arbitrary subset while extends declares a subset that is disjoint from any other extension of the same signature. If the parent signature is declared as abstract then it can only contain elements that are also contained in its extensions. The scope of run command only needs to define the number of elements that exist in top-level signatures, those that are not subsets of any other. For example, we could have the following variant of the above model.

sig Person {
  owns : set Car
}
sig Adult in Person {}
abstract sig Car {}
sig Sedan, SUV extends Car {}

In this model we have two disjoint extensions of abstract set Car, meaning all cars are either sedans or SUVs. We also have an arbitrary subset of Person that contains all the adult persons. If our requirement is again "Every person owns a car" the first positive instance above could be extended as follows, where no person is an Adult, Car1 is a Sedan and Car2 is a SUV.

run Instance1 {
  some disj Person1,Person2,Person3 : Person | some disj Car1, Car2 : Car {
    Person = Person1 + Person2 + Person3
  Adult = none
    Car = Car1 + Car2
  Sedan = Car1
  SUV = Car2
    owns = Person1->Car1 + Person2->Car2 + Person3->Car2
  }
} for 3 Person, 2 Car expect 1

A model can also declare relations of arity higher than 2 and use the module ordering to impose a total ordering on a signature. That total ordering is implicitly defined in a binary relation called next. For example ole following model includes the price of each car that is owned and Value is a totally ordered signature. Relation owns is now a ternary relation.

open util/ordering[Value]
sig Person {
  owns : Car -> lone Value
}
sig Car {}
sig Value {}

When specifying an instance for a model that uses ordering we must specify the value of the next total ordering, as done in the following instance. Since there can be multiple total orders you must disambiguate which next relation is being defined using the <: operator with the signature on the left. The same applies if more that one relation is declared with the same name.

run Instance1 {
    some disj Person1,Person2,Person3 : Person | some disj Car1, Car2 : Car | some disj Value1,Value2,Value3 : Value {
      Person = Person1 + Person2 + Person3
      Car = Car1 + Car2
  Value = Value1 + Value2 + Value3
      owns = Person1->Car1->Value1 + Person2->Car2->Value2 + Person3->Car2->Value2
  Value <: next = Value1->Value2 + Value2->Value3
  }
} for 3 Person, 2 Car, 3 Value expect 1

In this instance Car1 has the lowest price, since Value1 is the first element in the total order. Since owns is now a ternary relation its value is defined as sets of triples.

In the following interaction you will act as an Alloy expert. I will ask you to generate positive and negative instances for different requirements for a given model.

Here are some instructions:
- Do not attempt to formalize the requirement with a fact, just output the requested instances as run commands.
- I will clearly state how many positive and negative instances you should output.
- All instances should be truly different (be aware that the names of the elements are actually irrelevant).
- Try to produce minimal instances with few elements.
- All instances must include a comment explaining in natural language why it is positive or negative.
- All instances must define the scopes for all top-level signatures in the model.
- In every instance you must specify the values for all declared sets and relations with an equality restriction.
- To declare the elements of each signature use a some quantifier with the disj keyword to ensure that all elements are different.
- The value of a relation should take into account its arity. For example, if it is a ternary relation its value must be a set of triples.
- If a set or relation is empty you must explicitly state that it is equal to the empty set (none) or empty relation (of the correct arity).
- If there are two relations with the same name you must disambiguate using the <: operator with the domain signature on the left.
- If a model uses ordering on a signature you must also specify the value of the next binary relation using the <: operator to disambiguate. Signatures with ordering cannot be empty so next is never empty.
- If the value of some sets and relations is irrelevant for a given requirement you can just state that they are empty or assign them random values.
- Generate only Alloy code, do not include any explanations outside the code.
```

Finding 1

GPT-5 with a few-shot system prompt is highly effective at generating Alloy test suites for structural requirements expressed in natural language. It is also more cost-effective due to less reasoning effort.

Prompt	Tests	Syntax	Consistent	Previous	Valid	%	Cost
Few	258	255	255	252	247	96%	$3.56
One	258	226	208	206	205	79%	$3.69
Zero	258	137	120	119	118	46%	$4.20

Finding 2

The overall effectiveness of GPT-5 when using a few-shot prompt is not significantly affected by non-determinism.

Run	Tests	Syntax	Consistent	Previous	Valid	%	Cost
1st	258	255	255	252	247	96%	$3.56
2nd	258	256	256	256	251	97%	$3.50
3rd	258	252	252	250	246	95%	$3.61

Provider	Model
Open AI	GPT-5
DeepMind	Gemini 2.5 Pro
Anthropic	Claude Opus 4.1
Open AI	GPT-5 Mini
Meta	Llama 3.1 8B

Finding 3

Different LLMs struggle with different aspects of test case generation, some struggle more with syntax while others with semantics.

Model	Tests	Syntax	Consistent	Previous	Valid	%	Cost
GPT-5	258	255	255	252	247	96%	$3.56
Gemini	258	248	220	212	210	81%	$2.78
Claude	258	258	258	202	197	76%	$5.55
Mini	258	188	183	179	174	67%	$0.53

Finding 4

The syntactic errors made by GPT-5 are due to a very peculiar feature of Alloy syntax and easily fixable by some post-processing.

Finding 5

GPT-5 struggles more with negative test cases then with positive ones, but only on specific requirements that our anecdotal evidence suggests to be also ambiguous to humans.

Finding 6

GPT-5 generates diverse test cases for most requirements.

N	Complete	Wrong	Missed	Mean %
1	41	5587	2204	38.10%
2	34	4623	814	17.02%
3	36	4395	525	9.90%
4	35	5129	446	7.45%
5	35	4816	348	6.43%

Conclusion

GPT-5.5

Generate 5 positive and 5 negative instances for the requirement "Only students can be enrolled" for the following model. All instances must also satisfy the requirements "There are no people besides professors and students", "All courses have professors", "Only professors teach", and "No one teaches themselves”.

```
sig Person {
  teaches : set Course,
  enrolled : set Course
}
sig Professor in Person {}
sig Student in Person {}
sig Course {}
```

```
run PositiveInstance4 {
  /* Positive: the only enrolled person is Person1, who is a student.
     Person1 is also a professor but teaches Course1, while enrolled only in Course2. */
  some disj Person1, Professor2 : Person | some disj Course1, Course2 : Course {
    Person = Person1 + Professor2
    Professor = Person1 + Professor2
    Student = Person1
    Course = Course1 + Course2
    teaches = Person1->Course1 + Professor2->Course2
    enrolled = Person1->Course2
  }
} for 2 Person, 2 Course expect 1
```

Future work

- Can we use LLMs to also generate scenarios for validating behavioral requirements?
- Can we use open-source SLMs to make test generation more accessible?

Thanks!

This work is funded by national funds through FCT – Fundação para a Ciência e a Tecnologia, I.P.,
under the support UID/50014/2025 (https://doi.org/10.54499/UID/50014/2025).