Quick Start

CatSAT is a declarative programming system designed for procedural content generation in games. It allows you to define a generation *problem* by specifying the attributes a solution would have, and any *constraints* on the attributes. For example, if you are generating monsters the monster might be defined by its general body morphology, the kinds of attack(s) and defense(s) it has, any special weaknesses, etc. A constraint might be, for example, that a creature with a fire attack probably shouldn’t be weak against fire. A *solution* to the problem is a set of setting for the attributes that obeys or *satisfies* the constraints. A problem will generally have many solutions. CatSAT allows you to define problems and generate random solutions for them.

# Getting started

To use CatSAT, drop the CatSAT DLL into your Unity project. In the Unity editor, go into Player Settings and make sure that “Scripting Runtime Version” is set to “.NET 4.6 equivalent” or better. You cannot currently place the source files for CatSAT in your Unity project because the source uses C# 7, which Unity doesn’t yet support as of this writing. But the DLL targets .NET 4.6, so it will run fine in Unity 2018.

Then make a component that uses the namespace CatSAT:

using CatSAT;  
using static CatSAT.Language;

# Problems and Solutions

You start by making an instance of CatSAT.Problem:

var p = new Problem("test");

The argument is just a name to help you distinguish between problems if you have multiple problems in your game.

**Important:** creating a Problem, also sets a piece of global state, the current problem, which is stored in the static field Problem.Current. Many of the operations for declaring attributes and constraints implicitly define them for the current problem. CatSAT is written with the assumption that you will create a problem and fully specify it, before creating a second problem. That said, you can set Problem.Current to whatever problem you like if you need to go back and add things to a problem. You may not adding things to problems once you have solved them, however, since the solution process involves a preprocessing that’s difficult to undo.

You can now ask the problem for a Solution by saying:

var s = p.Solve();

The solution holds the settings for the different attributes of the problem. Since we haven’t specified any attributes, the solution isn’t terribly useful at the moment.

# Propositions

The simplest kinds of attributes, Propositions, are Boolean valued (true or false). Propositions belong to a particular Problem object, and are distinguished by a Name field, which can be any C# object. Two Propositions of the same Problem with the same Name must be the same object. You can cast an object to type Proposition to look up the proposition with that name in the current problem. If there is no such proposition, it will create one:

var p = new Problem("test");

var a = (Proposition)"a"; // Add propositions to p  
var b = (Proposition)"b";  
var c = (Proposition)"c";

The now when we solve the problem the solution will contain values for the propositions, and these can be accessed by using the solution’s indexer:

var s = p.Solve();

Debug.Log($"a={s[a]}, b={s[b]}, c={s[c]}");

# Adding constraints

Now let’s supposed we want to generate a monster. The monster will have a body type of bird, fish, or mammal:

var p = new Problem("monster");

var bird = (Proposition)"bird";  
var fish = (Proposition)"fish";  
var mammal = (Proposition)"mammal";

If we ask for a solution now, the different body types will be completely unrelated. The system can choose to make the monster a bird, a bird and a fish at the same time, nothing, etc. We want to add the constraint that a monster has to be exactly one of these. We can do this by adding a uniqueness constraint:

p.Unique(bird, fish, mammal);

The system will now only generate configurations in which exactly one body type is selected. You might reasonably object that it would be better to have one body type variable with the possible values of bird, fish, and mammal, rather than three constrained Boolean variables. That’s called “non-Boolean SAT”, and we’ll talk about that later. For the moment, let’s stick with the Boolean version.

Now let’s add a few attack types and assume for the moment that the monster can only have one form of attack:

var bite = (Proposition)"bite";  
var claw = (Proposition)"claw";  
var fire = (Proposition)"fire breathing";  
p.Unique(bite, claw, fire);

Unfortunately, we now have the situation in which we can have fire-breathing fish, which doesn’t make a lot of sense. So we can say those are inconsistent:

p.Inconsistent(fish, claw);  
p.Inconsistent(fish, fire);

Alternatively, since biting is the only option left for fish, we could have told it that directly:

p.Assert(fish > bite);

Here the “>” is supposed to be read as looking like the symbol used in logic. So this means , i.e. “fish implies bite”. Another way of saying this would be:

p.Inconsistent(fish, Not(bite));

which ultimately means the same thing.

**Note:** from a software engineering standpoint, it’s probably best to use our original list of two explicit Inconsistent declarations. We’re likely to add new kinds of attacks in the future; the > and Not versions will both rule out those new attacks for fish, while the original version will not.

# Quantification

Unique and inconsistent are examples “quantification” constraints: they specify how many of a set of propositions can be concurrently true or false. Unique says exactly one must be true, while inconsistent says that at least one must be false. There are a number of other quantifiers:

* *p*.All(*propositions*, …)  
  They all have to be true
* *p*.AtLeast(*number*, *propositions*, …)  
  At least *number* must be true
* *p*.AtMost(*number*, *propositions*, …)  
  At most *number* must be true
* *p*.Exactly(*number*, *propositions*, …)  
  Exactly *number* must be true
* *p*.Quantify(*min*, *max*, *propositions*, …)  
  Between *min* and *max* must be true.

These constraints also allow you to pass a single IEnumerable<Proposition> as an argument, in lieu of the multiple proposition arguments, or an IEnumerable<T> and a Func<T,Proposition> argument.

# Rules

The last important class of constraint is, for want of a better name, called a Rule. Rules look like implications, except that they’re written *a* <= *b* rather than *b* > *a*, i.e. the conclusion is on the left rather than the right. This is to make sure they look very different, since they have subtly different semantics.

The semantic difference is that > is a one-way constraint, but <= is a bi-directional constraint. The implication *b* > *a* means only that *b* being true makes *a* true; *a* can be true without *b* being true. However, if we have a set of rules:

p.Assert(a <= b, a <= c, a <= d);

then if any of *b*, *c*, *d* are true, that will force *a* to be true also. However, it also means that if *a* is true, then at least one of *b*, *c*, *d* must be true. A rule adds the constraint that its conclusion is true *if and only if* some rule justifies it.

So for example, if we say:

p.Assert(shot > dead,  
 drowned > dead,  
 fallen\_off\_cliff > dead);

This will allow a solution in which the person is dead, without having been shot, drowned, or fallen off a cliff. The can be dead without any particular inference that supports their being dead. On the other hand, if we say:

p.Assert(dead <= shot,  
 dead <= drowned,  
 dead <= fallen\_off\_cliff);

then they can’t be dead without also having been shot, drowned, or fallen off a cliff.

Here’s another, less violent example:

p.Assert(air\_breather <= mammal,  
 air\_breather <= bird);

This says that air\_breather is true for all birds and mammals, and only for them.

# Boolean operators

You can also use the operators & and Not on the right hand sides of rules, e.g.:

p.Assert(dead <= (gun\_loaded & shot & Not(shot\_blocked)));

Note the parentheses around the right-hand side, which are required by C#’s precedence rules. You can also use Boolean operators on the left hand sides of > implications.

# Forcibly setting values of propositions

You can also specify that you want a particular proposition to be true or false. If we say:

p[dead] = false;

var s1 = p.Solve();

Then we will get a solution back in which dead will be false. That will necessarily mean that shot, drown, and fallen\_off\_cliff are also false. However, if we then say:

p[dead] = true;

var s2 = p.Solve();

then the new solution will be guaranteed to have dead true, and so at least one of shot, drowned, or fallen\_off\_cliff, must also be true. You can remove the value assigned to dead by saying:

p.ResetProposition(dead);

This will tell the system to choose dead randomly again. To remove all values assigned to all propositions, use:

p.ResetPropositions();

# Predicates

A predicate is essentially a proposition that can take arguments. In CatSAT, these are implemented as C# functions (delegates) that return Propositions. So a proposition that takes two numbers as arguments has C# type Func<int, int, Proposition>. For convenience, CatSAT provides the generic static method Predicate (in CatSAT.Language) to make predicates.

Suppose we wanted to rebuild our monster generator to be able to make parties of monsters. Then propositions like bite and claw, would need to be predicates that took the monster as an argument and returned the proposition that that monster bit or clawed, respectively. We can do this in CatSAT with this:

var p = new Problem("party");

var bird = Predicate<string>("bird");  
var fish = Predicate<string>("fish");  
var mammal = Predicate<string>("mammal");

var bite = Predicate<string>("bite");  
var claw = Predicate<string>("claw");  
var fire = Predicate<string>("fire breathing");

Bite, claw, etc. are now predicates that take the (string) name of the monster and return a unique Proposition corresponding to that monster biting, clawing, etc. However, now we can no longer say:

p.Unique(bird, fish, mammal);  
p.Unique(bite, claw, fire);

because bird, fish, bite, and so on aren’t propositions. We have to say, instead, that these are true for all members of the monster party. That means we have to decide in advance how many monsters are in the party and what their names are. Then we iterate over all the members of the party and assert their uniqueness in turn:

var party = new string[] {"groucho", "harpo", "chico" };

foreach (var who in party) {  
 p.Unique(bird(who), fish(who), mammal(who));  
 p.Unique(bite(who), claw(who), fire(who));  
}

Note that, we didn’t need to name the monsters, we could have just numbered them:

var p = new Problem("party");

var bird = Predicate<int>("bird");  
var fish = Predicate<int>("fish");  
var mammal = Predicate<int>("mammal");

var bite = Predicate<int>("bite");  
var claw = Predicate<int>("claw");  
var fire = Predicate<int>("fire breathing");  
  
int partySize = 3;  
foreach (var who in Enumerable.Range(1,partySize)) {  
 p.Unique(bird(who), fish(who), mammal(who));  
 p.Unique(bite(who), claw(who), fire(who));  
}

Now let’s say that we don’t want all our monsters to be birds. We can do that by saying:

p.AtMost(2, bird(1), bird(2), bird(3));

Or, more scalably:

p.AtMost(2, Enumerable.Range(1,partySize), who => bird(who));

This version takes an enumerator as a second argument and a function to map the enumerated elements to propositions as the third argument. This can also be simplified to:

p.AtMost(2, Enumerable.Range(1,partySize), bird);

Or:

var party = Enumerable.Range(1,partySize);  
p.AtMost(2, party, bird);

We can require there is at least one bird by changing AtMost to AtLeast. In this way, we can place whatever limits on party composition that we like.

# Non-Boolean SAT (experimental)

Finally, it’s possible to make kinds of variables that aren’t propositions. The API for this is less stable, but the quickstart version is to say:

var bodyType = new FDVariable("body type",  
 "bird", "mammal", "fish");

where the first argument is a string name for the variable (for debugging purposes) and the subsequent arguments are its possible values, which may be of any type (FDVariable is a generic type). Solutions will now include a value for bodyType by asking it for its value in a given solution:

var s = p.Solve();  
Debug.Log(bodyType.Value(s));

Unfortunately, C# doesn’t allow us to define generic indexers, so you can’t get its value by saying s[bodyType].

Given an FDVariable, you can test its value using ==

var s = p.Solve();

if (s[bodyType == "bird"])  
 Debug.Log("It’s a bird!");

Note that technically, what’s going on here is that == is overloaded to return a Proposition. You can use that Proposition as you would use any other proposition, e.g.:

p.Inconsistent(bodyType == "fish", attackType == "claw");  
p.Inconsistent(bodyType == "fish", attackType == "fire");

There are a lot of other kinds of fancy things you can do, such as creating variables that are only defined in a solution under certain conditions, or using Reification to make reasonably normal-looking C# classes that are filled in by CatSAT (this has the advantage of allowing maximal type checking at compile time). They will get documented once their APIs stabilize. If you want to use them in the meantime, feel free to drop me a note at [ian@northwestern.edu](mailto:ian@northwestern.edu).

**Note:** FDVariable stands for “finite-domain variable” meaning its domain (its set of possible values) is a finite set. There are other kinds of variables, including FloatVariables, but these are even more experimental.