Tutorial

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 collection of settings 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.x 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;

You will also probably want to add someplace in your code the call:

CatSAT.Random.SetSeed();

which will initialize CatSAT’s RNG seed based on the time.

# 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 add 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 suppose 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 mammal, a bird and a fish at the same time, none of them, etc. We want to add the constraint that a monster 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(bite < fish);

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 several 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 *a* < *b*.

The semantic difference is that < is a one-way constraint, but <= is a two-way constraint. The implication *a* < *b* means only that *b* makes *a* true; but *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, as with <. 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(dead < shot,  
 dead < drowned,  
 dead < fallen\_off\_cliff);

This will allow a solution in which the person is dead, without having been shot, drowned, or fallen off a cliff. They 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 always get a solution 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 Proposition objects. So, a predicate 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 must 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.

## Specialized predicates

You can also define symmetric predicates using the SymmetricPredicate function. You can also use any Boolean-valued function inside a rule as if it were a predicate.

# Non-Boolean SAT

Finally, it’s possible to make kinds of variables that aren’t propositions. The short version is to say:

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

var attackType = new FDVariable("attackType",  
 "bite", "claw", "fire");

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(s[bodyType]);

Note that s[bodyType] returns type object because there are many different kinds of variables with different kinds of values. So if you’re doing anything other than printing it, you need to cast it to string. You can avoid this by saying bodyType.Value(s), which will be properly typed to string.

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");

**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.

## Conditional variables

Some variables don’t make sense except under certain conditions. For example, let’s say that some fire-breathers have long-range attacks and others have short range attacks. We could add a range variable:

var range = new FDVariable("range",  
 "long", "medium", "short");

But now, we have monsters biting monsters that are “long range” biters, which is a little odd. So let’s tell the system that the range variable is only defined (only has a value) for fire breathers. We do this by adding an extra constructor argument, the variable’s *condition*, which is a proposition that must be true for the variable to have a value:

var range = new FDVariable("range", attackType == "fire",  
 "long", "medium", "short");

Now the range variable will always have a unique value for fire breathers, and no value for other kinds of monsters.

# Getting data out of CatSAT

Having obtained a solution, you can get the value of any particular proposition or variable, as we did above, by using the solution’s indexers, *solution*[*proposition*] and *solution*[*variable*]. Alternatively, you can copy all the data out of a solution object into some other C# object using the Populate() method. Populate() uses reification to look of the fields of an object, find the variable or proposition from the solution with the same name, and set the C# member to the value of the associated CatSAT variable. So if we make a Monster class, with fields for body type and so on:

class Monster {  
 public string bodyType;  
 public string attackType;  
 public string range;  
}

Now you can randomly generate a valid Monster object using CatSAT by saying (assuming s is the solution object from above):

var m = new Monster();  
s.Populate(m);

This will fill in the fields of m with their values from the solution. Better yet, we can move the code for defining the problem object inside the monster class:

class Monster {  
 public string bodyType;  
 public string attackType;  
 public string range;  
  
 static Problem Problem;  
 static Monster() {  
 Problem = new Problem("Monster generator");  
 … *code from above* …  
 }

public Monster() {  
 Problem.Populate(this);  
 }  
}

We now have a C# class that automatically generates its own instances.

This code has the disadvantage that the fields are all strings. We’d rather have them be enumerated types. We can do that easily. Just define our enumerations:

enum AttackType { Bite, Claw, Fire }  
enum BodyType { Fish, Mammal, Bird }  
enum RangeType { Long, Medium, Short }

And now change our variables from FDVariable<string> to EnumVariable<AttackType>, etc. EnumVariables are like FDVariables, but they know to get their possible values from the specific enumeration type.

# Examples

The appendices include extended code examples:

* *AutoDread*:an example of using CatSAT to constraint choices in a user interface
* *PCGToy*: this is a simple editor for generators that I made originally for PROCJAM.
* *Storyteller demo*: an example of more complicated inference
* A series of variants of the same attribute generator, showing different ways of coding it depending on the developer’s needs
* A lineage generator for random extended families. This shows an example of how to generate trees in a SAT language.

# Example: AutoDread

This is in a separate github repo in my account.

This is an implementation of the character questionnaire from the TTRPG *Dread*. It’s the beginnings of an implementation of a *Dread*-like game in Unity. Right now, we just have the character questionnaire mechanic. It administers a series of questions to you and lets you choose from answers for each. It filters answers that contradict previous choices, and questions that are contradicted by or fully determined by previous answers. It also shows an example of using a custom DSL to encode the SAT problem. Apologies for the lame parser that it uses.

# Example: PCGToy

This is a simple editor for generators that I made originally for PROCJAM. It runs on Windows Forms and is another example of implementing a custom DSL for describing your SAT problems. It’s intended as an aide for GMs running table-top RPGs.

I’ve also included a loader for its files as a separate Visual Studio project, and a proof-of-concept Unity project that uses a PCGToy file to configure unity components. To try it out, clone the repo, compile the visual studio solution, and put copies of the CatSAT and PCGToyLoader DLLs in the Plugins directory of the unity project. The inspector will initially show the fields of the different Merchant objects as null. Now click the run button to start the game and use the inspector to look at the different Merchant objects; you will see that they’ve all been randomly autogenerated. Look at Merchant.scm, or open it in the PCGToy app, to see what the domains, variables, and constraints are that are defined in it.

The PCGToyUnity project is a quick and dirty thing I threw together to demonstrate the kind of thing one can do. It’s not intended to be a finished product ready for people to use in their games. For example, it pretty much only understands strings right now. It’s conceptually easy to extend it to support other kinds of variables. However, the design space is sufficiently under-constrained that I’d rather get input from a user who wants to use it in a game before adding features. (To understand why, just take a look at the example below called “the same generator written too many ways.”)

# Example: Reimplementing the demo from *Storyteller*

*Storyteller* is a forthcoming narrative puzzle game by Daniel Benmergui. In it, there are three characters who appear in three comic-book panels. The player can change the positions of the characters in the first two panels, and the last panel shows the outcome. The first panel shows each character either next to a castle or a hovel. Poverty makes a character evil, and the second panel shows the evil character taking at least one of the other characters captive, with the third character either also taken captive or fighting the evil character. The final panel shows who lives: the evil character if no one fights her/him, the others, if they do. This a reverse-engineered axiomatization of the game developed by the students in EECS-396 as part of an in-class exercise, and which I then translated into CatSAT:

var p = new Problem("Storyteller demo rebuild");

// Characters identified by color

var cast = new[] {"red", "green", "blue"};

// Character is rich in the first panel

var rich = Predicate<string>("rich");

// Character is a prisoner in the second panel

var caged = Predicate<string>("caged");

// Character has a sword in the second panel

var hasSword = Predicate<string>("hasSword");

// Character is evil

var evil = Predicate<string>("evil");

// First argument character stabbed the second

var stabbed = Predicate<string, string>("stabbed");

// First argument character loves the second

var loves = Predicate<string, string>("loves");

// A tombstone is displayed for the character at the end

var tombstone = Predicate<string>("tombstone");

// Some non-evil character is free in the second panel

var someoneFree = (Proposition) "someoneFree";

// Panel 1 -> panel 2

foreach (var x in cast)

{

p.Assert(  
 // Povert causes evil in this world

evil(x) == Not(rich(x)),

// Only rich people are taken prisoner

caged(x) > rich(x),

// You have a sword iff your right and uncaged

hasSword(x) == (rich(x) & Not(caged(x))),

// If someone’s not caged, someone’s free

someoneFree <= Not(caged(x)),

// No suicide

Not(stabbed(x,x))

);

// You can't kill multiple people

p.AtMost(1, cast, y => stabbed(x, y));

foreach (var y in cast)

p.Assert(

// You need a sword to stab

hasSword(x) < stabbed(x, y),

// Only stab evil people

evil(y) < stabbed(x,y)

);

}

// Panel 2 -> panel 3

foreach (var x in cast)

{

foreach (var y in cast)

{

p.Assert(

// Dead iff you’re stabbed, or caged and not rescued

tombstone(x) <= (caged(x) & evil(y) & Not(dead(y))),

tombstone(x) <= Not(someoneFree),

tombstone(x) <= stabbed(x, y)

);

foreach (var z in cast)  
 // You love someone if they rescue you

p.Assert(loves(x, y) <= (caged(x) & stabbed(y, z)));

}

}

# Example: The same generator, written too many times

This isn’t a super exciting example, but suppose we want to generate parties of NPCs. A character has a race and a class. Races are human, electrode, and insectoid. And classes are fighter, magic user, cleric and thief. Humans and clerics each have one extra attribute. Humans have a nationality (Landian, Placevillian, or Cityburgher), and clerics have a religion (monotheist, pantheist, Lovecraftian, or Dawkinsian). And let’s add some constraints (otherwise, there’s no need to use a constraint solver!):

* Lovecraftianism is the state religion of Cityburgh
* Lovecraftianism is illegal in Landia
* Insectoids are always pantheists

## Making a character using propositions

Let’s start by making just one character in the simplest way possible. We’ll have a set of propositions to represent being human, being a cleric, etc. Remember that there’s an implicit cast from strings to propositions, so we can just write "human" instead of (Proposition)"human":

var p = new Problem("Character generator");

p.Assert("character");

// Races: characters are divided into human, electrode, and insectoid

Partition("character", "human", "electroid", "insectoid");

// Classes: character also divided into fighter, etc.

Partition("character", "fighter", "magic user", "cleric", "thief");

p.Inconsistent("electroid", "cleric");

// Nationalities: humans are divided into Landian, etc.

Partition("human", "landia", "placeville", "cityburgh");

// Religions: clerics are divided into monotheist, etc.

Partition("cleric", "monotheist", "pantheist", "lovecraftian", "dawkinsian");

// Constraints

p.Inconsistent("landia", "lovecraftian");

p.Inconsistent("insectoid", "pantheist");

p.Inconsistent("cityburgh", "cleric", Not("lovecraftian"));

where Partition(a, b, c, d) means:

* If a is true, then one of b, c, or d must be true
* But only one of b, c, and d
* And b, c, and d can only be true if a is true.

The definition of Partition is a little ugly (sorry):

void Partition(Proposition prop, params Proposition[] subprops)

{

// subprop implies prop

foreach (var subprop in subprops)

Problem.Current.Assert(prop < subprop);

// Can’t have prop true and all the subprops false

// Apologies for my possibly excessive devotion to functional programming

Problem.Current.Inconsistent(subprops.Select(Not).Concat(new[]{prop}));

// The subprops are mutually exclusive

Problem.Current.AtMost(1, subprops);

}

## Making a party using predicates

Now let’s make a party of characters. To do that, we need to abstract the code above by changing the propositions to predicates that map a character name to the relevant proposition for that particular character:

var p = new Problem("Party generator");

var character = Predicate<string>("character");

var human = Predicate<string>("human");

var electroid = Predicate<string>("electroid");

var insectoid = Predicate<string>("insectoid");

var fighter = Predicate<string>("fighter");

var magicUser = Predicate<string>("magic\_user");

var cleric = Predicate<string>("cleric");

var thief = Predicate<string>("thief");

var landia = Predicate<string>("landia");

var placeville = Predicate<string>("placeville");

var cityburgh = Predicate<string>("cityburgh");

var monotheist = Predicate<string>("monotheist");

var pantheist = Predicate<string>("pantheist");

var lovecraftian = Predicate<string>("lovecraftian");

var dawkinsian = Predicate<string>("dawkinsian");

Now we assert the constraints for every member of the party:

var party = new[] {"fred", "jenny", "sally"};

foreach (var who in party)

{

p.Assert(character(who));

Partition(character(who), human(who), electroid(who), insectoid(who));

Partition(character(who),  
 fighter(who), magicUser(who), cleric(who), thief(who));

p.Inconsistent(electroid(who), cleric(who));

Partition(human(who), landia(who), placeville(who), cityburgh(who));

Partition(cleric(who),  
 monotheist(who), pantheist(who), lovecraftian(who),   
 dawkinsian(who));

p.Inconsistent(landia(who), lovecraftian(who));

p.Inconsistent(insectoid(who), pantheist(who));

p.Inconsistent(cityburgh(who), cleric(who), Not(lovecraftian(who)));

}

Finally, let’s add the constraint that the different members of the party should have different classes:

p.AtMost(1, party, fighter);

p.AtMost(1, party, magicUser);

p.AtMost(1, party, cleric);

p.AtMost(1, party, thief);

These say: for each element of party, call fighter (or magicUser, etc.) on it, collect all the resulting propositions, and assert that at most 1 of those can be true. It’s equivalent to:

p.AtMost(1, fighter("fred"), fighter("jenny"), fighter("sally"));

## One character using variables

Using many distinct propositions to represent each race and class is ugly. So now let’s write it using just one variable each for race and class.

var p = new Problem("Character generator");

var race = new FDVariable<string>("race", "human", "electroid", "insectoid");

// Have to call this cclass since class is a reserved word in C#

var cclass = new FDVariable<string>("class",  
 "fighter", "magic user", "cleric", "thief");

p.Inconsistent(race == "electroid", cclass == "cleric");

// Nationality is only defined when race == "human"

var nationality = new FDVariable<string>("nationality", race == "human",

"landia", "placeville", "cityburgh");

// Religion is only defined when cclass == "cleric"

var religion = new FDVariable<string>("religion", cclass == "cleric",

"monotheist", "pantheist", "lovecraftian",   
 "dawkinsian");

// Constraints

p.Inconsistent(nationality == "landia", religion == "lovecraftian");

p.Inconsistent(race == "insectoid", religion == "pantheist");

p.Inconsistent(nationality == "cityburgh", cclass == "cleric",  
 Not(religion == "lovecraftian"));

Now we can ask a solution for the value of the cclass variable, rather than having to test each possible class proposition in turn.

## Generating one character using one struct variable

So now instead of representing a character as a bunch of propositions, we’ve represented it as four variables: race, class, nationality, and religion. But we still don’t have a data object that represents the character. We can do that with a different kind of variable, as structure:

var characterType = new Struct("Character",  
 // Member variables of a Struct variable

new []{

new Member("race", null, "human", "electroid", "insectoid"),

new Member("class", null, "fighter", "magic user", "cleric", "thief"),

new Member("nationality", "race=human",  
 "landia", "placeville", "cityburgh"),

new Member("religion", "class=cleric",  
 "monotheist", "pantheist", "lovecraftian", "dawkinsian")

},  
 // Constraints to apply to instances of this struct  
 // p is the problem we’re adding it to.

// v is a dictionary of the different member variables

(p, v) =>

{

p.Inconsistent(v["race"] == "electroid", v["class"] == "cleric");

p.Inconsistent(v["nationality"] == "landia",  
 v["religion"] == "lovecraftian");

p.Inconsistent(v["race"] == "insectoid", v["religion"] == "pantheist");

p.Inconsistent(v["nationality"] == "cityburgh", v["class"] == "cleric",   
 v["religion"] != "lovecraftian");

});

Now we can add all the variables for a character to a problem by instantiating the type

var p = new Problem("Struct character generator");

p.Instantiate("character", characterType);

## Generating a party using struct variables

Now we can make a party by instantiating the struct repeatedly:

var p = new Problem("Struct character generator");  
var fred = p.Instantiate("fred", characterType);  
var jenny = p.Instantiate("jenny", characterType);  
var sally = p.Instantiate("sally", characterType);

Now Fred, Jenny, and Sally are each struct variables (variables of type CatSAT.StructVar) containing their own race, class, nationality, and religion. Alternatively, we could write functionally as:

var party = new[] { "fred", "jenny", "sally" };

var partyVars = party.Select(  
 c => (StructVar)p.Instantiate(c, characterType)).ToArray();

Finally, we need to add the constraint that different characters have different classes:

p.AllDifferent(partyVars.Select(c => (FDVariable<string>)c["class"]));

The Select call iterates over all the partyVars c, looks up the variable named "class" inside it, and casts it to be an FDVariable<string>, since c is allowed to include variables of different types. The AllDifferent call then requires that all the variables have different values; it just compiles into a bunch of AtMost calls.

## Making characters be real C# objects

Now you might object that characters aren’t real C# objects. You want to be able to say fred.race, not fred["race"]. You can do that with the CompiledStruct class, which lets you make user-defined classes that also function as StructVars. The constructor for the class is called with the problem in which to instantiate the variable, as well as the condition on which this variable should be defined (ignore that for now), and then it adds whatever constraints it needs to. The members of the class are then added to the problem automatically. Their domains are specified using a Domain attribute, and those fields whose values are only defined under specified conditions are specified using the Condition attribute:

class Character : CompiledStruct

{

public Character(object name, Problem p, Literal condition = null)  
 : base(name, p, condition)

{

p.Inconsistent(Race == "electroid", Class == "cleric");

p.Inconsistent(Nationality == "landia", Religion == "lovecraftian");

p.Inconsistent(Race == "insectoid", Religion == "pantheist");

p.Inconsistent(Nationality == "cityburgh", Class == "cleric",  
 Religion != "lovecraftian");

}

[Domain("race", "human", "electroid", "insectoid")]

public readonly FDVariable<string> Race;

[Domain("class", "fighter", "magic user", "cleric", "thief")]

public readonly FDVariable<string> Class;

[Domain("nationality", "landia", "placeville", "cityburgh"),  
 Condition("Race", "human")]

public readonly FDVariable<string> Nationality;

[Domain("religion", "monotheist", "pantheist", "lovecraftian", "dawkinsian"),   
 Condition("Class", "cleric")]

public readonly FDVariable<string> Religion;

}

Now we can make a party of these characters and we can access their class just by saying .Class rather than by saying ["class"]:

var p = new Problem("compiled struct party generator");

var party = new[] { "fred", "jenny", "sally" };

// Make one for each party member

var t = new CompiledStructType(typeof(Character));

var partyVars = party.Select(c => (Character)p.Instantiate(c, t)).ToArray();

// All the classes have to be different

p.AllDifferent(partyVars.Select(c => c.Class));

## Getting rid of the strings

Finally, you might object that the values of the variables are represented as strings. That means that if you mistype a string someplace, there’s no compile-time checking. You might prefer to have the values of the variables be enumerated types rather than strings. To do that, we just define our enum types and replace the FDVariables in the class with EnumVariables. Otherwise the code is identical:

enum Races { **Human**, **Electroid**, **Insectoid** }

enum Classes { Fighter, MagicUser, **Cleric**, Thief }

enum Nationalities { **Landia**, Placeville, **Cityburgh** }

enum Religions { Monotheist, **Pantheist**, **Lovecraftian**, Dawkinsian }

class CharacterWithEnums : CompiledStruct

{

public CharacterWithEnums(object name, Problem p, Literal condition = null)  
 : base(name, p, condition)

{

p.Inconsistent(Race == Races.**Electroid**, Class == Classes.**Cleric**);

p.Inconsistent(Nationality == Nationalities.**Landia**,  
 Religion == Religions.**Lovecraftian**);

p.Inconsistent(Race == Races.**Insectoid**, Religion == Religions.**Pantheist**);

p.Inconsistent(Nationality == Nationalities.**Cityburgh**,  
 Class == Classes.**Cleric**,  
 Religion != Religions.**Lovecraftian**);

}

public readonly EnumVariable<Races> Race;

public readonly EnumVariable<Classes> Class;

[Condition("Race", Races.**Human**)]

public readonly EnumVariable<Nationalities> Nationality;

[Condition("Class", Classes.**Cleric**)]

public readonly EnumVariable<Religions> Religion;

}

Now there are no more strings, so we now get compile-time checking and IDE support for autocompletion, etc.

## Making it data-driven

You may well prefer to specify the variables, values, and constraints in a separate text file. To do that, take a look at the PCGToy project in the CatSAT repo. It’s a GUI for defining and editing problems like this. You might also look at the repo for *AutoDread*, an implementation of the questionnaire mechanic from the game *Dread*, using CatSAT.

# Example: a family tree generator

This is a routine that generates a random family tree for a number of generations of a family. It’s very simple and *extremely* heternormative: it doesn’t understand gay couples, non-binary genders, adoption, divorce, or even the possibility of a someone *not* marrying a member of the opposite sex. Nor does it attempt to assign attributes to the family members, other than their sex. Characters don’t even have names, just numbers. But it’s a base you can work from to add whatever other capabilities you like.

Here’s the core idea. The family members are numbered , and with the mate of character always being character number . Assuming there is an incest taboo, the mate of someone within the family will always be someone from outside of the family. We can represent the family tree as follows:

* Character 1 is the matriarch, with character -1 being the patriarch. They are the root of the tree.
* Characters with positive numbers are descendants of the matriarch. Characters with negative numbers are characters from outside the family marrying in.
* The family line (the matriarch and her descendants) are therefore the characters .
* Characters and are always married and of opposite sexes
* For any members of the family line, and , is true iff is the parent of from within the family line. The other parent is, by definition, and is from outside the family line.
* , states that character is of generation . The matriarch is generation 1.

We then apply the following constraints:

* Everyone in the family line except the matriarch has a unique parent in the family line  
  this forces the tree to be well-formed.
* Parents are of opposite sexes
* The family line is approximately sex balanced (between 40% and 60% female). Otherwise you can and do get trees that are wildly unbalanced.
* The generation of someone is always one larger than that of their parent:  
  This prevents generating cyclic parentage graphs
* Every generation from has at least one member within the family line  
  This enforces the tree having a specified depth.

Here’s the code:

var p = new Problem("family generator");

var FamilySize = 25;

var generationCount = 5;

var matriarch = 1;

var familyLine = Enumerable.Range(matriarch, FamilySize).ToArray();

var kids = Enumerable.Range(matriarch+1, FamilySize-1).ToArray();

var childGenerations = Enumerable.Range(1, generationCount - 1).ToArray();

var female = Predicate<int>("female");

var generation = Predicate<int, int>("generation");

var parent = Predicate<int, int>("parent");

// Make of person # who is person number -who

int Mate(int who) => -who;

// Family must be 40-60% female

p.Quantify((int)(FamilySize\*.4), (int)(FamilySize\*.6), kids, female);

// Matriarch is the generation 0 female

p.Assert(female(matriarch), Not(female(Mate(matriarch))));

p.Assert(generation(matriarch, 0));

foreach (var child in kids)

{

// Everyone has exactly one parent from within the family line

p.Unique(familyLine, par => parent(child, par));

// Everyone has a generation number

p.Unique(childGenerations, g => generation(child, g));

p.Assert(

// Only matriarch and patriarch are generation 0

Not(generation(child, 0)),

// Heteronormativity

female(child) == Not(female(Mate(child))));

foreach (var par in familyLine)

foreach (var g in childGenerations)

// Child's generation is one more than parent's generation

p.Assert(generation(child, g) <=   
 (parent(child, par) & generation(par, g-1)));

}

// Every generation has at least one kid

foreach (var g in childGenerations)

p.Exists(kids, k => generation(k, g));