42

Building Custom Static Checkers Using Declarative Programming

Ian Horswill, Robert Zubek, and Matthew Viglione

42.1 Introduction

Programmers have developed a wide range of tools for sanity checking their code. Compilers perform type checking and generate warnings about undefined or unused variables. Programs such as Microsoft's FxCop can detect opportunities for optimization, likely bugs such as files that are opened but never closed, and the use of unlocalized strings. These are all examples of *static* checking, as opposed to dynamic checks (such as assertions) that can only verify behavior for a particular run of the program.

Unfortunately, code is only a small part of a game. The rest of the content is composed of a diverse collection of media objects, developed using an equally diverse collection of tools. Many of those tools are purpose-built, whereas others were never specifically designed to operate with one another. As a result, integrity checking of this noncode content, such as:

- Whether a referenced sound file actually exists
- Whether a tag string on an asset is a typo or a valid tag
- Whether all texture files are in the correct format is often limited

As a result, these problems are often only found when the problematic asset is used at runtime, resulting in the need for extensive manual testing and retesting.

The barrier to static checking of assets is economic, not technical. Unlike a compiler, whose costs can be amortized over thousands of programmers, the tool chain of a given game is used only by a few people, especially for indie titles. It does not make sense to invest thousands of person-hours in developing a tool that will save only hundreds of person-hours.

Fortunately, writing a static checker does not have to be expensive. In this chapter, we will show you how you can use declarative programming to build your own custom static checkers quickly and easily using a Prolog interpreter that can run inside of Unity3D. This will allow you to write a static checker as a series of statements that literally say "it is a problem if …" and let Prolog do the searching to find instances of the problem.

We have used this technique in two games, a research game, *MKULTRA*, and a commercial game, SomaSim's *Project Highrise*. In both cases, it has saved considerable effort tracking down bugs.

42.2 A Very Brief Introduction to Prolog

We will start by giving you a very brief introduction to Prolog. This will be enough to be able to read the code in the rest of the chapter, but if you want to write Prolog code, then it is worth reading a Prolog book such as *Prolog Programming for Artificial Intelligence* (Bratko 2012) or the first few chapters of *Prolog and Natural Language Analysis* (Pereira and Shieber 2002).

42.2.1 Prolog as a Database

At its simplest, Prolog acts as a fancy database. The simplest Prolog program consists of a series of "facts," each of which asserts that some predicate (relation) holds of some arguments. The fact:

```
age(john, 25).
```

asserts that the `age` relation holds between `john` and the number 25, `that is`, John's age is 25. Note that the period is mandatory; it is like the semicolon at the end of a statement in C. So you can write a simple relational database by writing one predicate per table, and one fact per row, in the database:

```
age(john, 25).
age(mary, 26).
age(kumiko, 25).
…
```

42.2.2 Querying the Database

Having added a set of facts about the ages of different people, we can then query the database by typing a pattern at the Prolog prompt, which is usually printed as `?-`. A pattern is simply a fact that can optionally include variables in it. Variables, which are notated by

capitalizing them, act as wildcards, and their values are reported back from the match. So if we ask who is age 26, Prolog will reply with Mary:

```
?- age(Person, 26).
Person=mary.
?-
```

Conversely, if we type `age(kumiko, Age)`, we will get back `Age=25`. We can include multiple variables in a query, and Prolog will report back all the different facts that match it. So if we type `age(Person, Age)`, Prolog will report all the `Person/Age` combinations found in the database. However, if a variable appears twice in a query, it has to match the same value in both places. So if we gave a query like `age(X, X)`, it would fail (produce no answers) because there are no `age` facts in the database that have the same value for both their arguments. Queries can also contain multiple predicates, separated by commas, such as in the query:

```
?- age(P1, Age), age(P2, Age).
```

(Remember that the `?-` is the prompt.) Here, the query asks for two people, `P1` and `P2`, who share the same `Age`. Although we might be expecting to get the reply `P1=john`, `Age=25`, `P2=kumiko`, the first reply we will actually get is `P1=john`, `P2=john`, since we never told it `P1` and `P2` should be distinct. We can revise the query to use Prolog's built-in not-equal predicate, `\=`, to get the meaning we intend:

```
?- age(P1, Age), age(P2, Age), P1 \= P2.
```

Prolog has most of the built-in comparisons, you would expect: =, <, >, *etc*. The only thing that is unusual about them is that "not" is notated with \ rather than ! as in C. So not equal is \= instead of != and not *X* is \+ *X* rather than !*X* as in C.

42.2.3 Rules

Prolog also allows you to package a query as a predicate. This is done by providing rules of the form *predicate*`:-` *query* to your program, for example:

```
same_age(P1, P2):- age(P1, Age), age(P2, Age), P1 \= P2.
```

Again, note the period at the end. This says that `P1` and `P2` are the same age if `P1`'s age is `Age`, `P2`'s is also `Age`, and `P1` and `P2` are different people. You can provide multiple rules and facts for a predicate, and Prolog will test them against your query in the order it finds them in your source file. You can now use `same_age` in your code as if it were a "real" predicate defined by facts, and Prolog will automatically handle queries against it by running the appropriate subqueries.

42.2.4 Prolog as Logical Deduction

So far, we have described Prolog as a kind of database. But we can also think of it as doing inference. Under that interpretation, the program is a set of premises: the facts are just

that—facts about what is true; and the *P*`:-` *Q* rules are logical implications: *P* is true if *Q* is true. The `:-` operator is logic's implication operator ⇒ (albeit with the arguments reversed), comma is "and" (∧), semicolon is "or" (∨), and `\+` is "not" (¬).

A query is a request for Prolog to prove there exists a set of values for the variables that make the query a true statement. Prolog does that by doing a depth-first search of the database to find a specific set of values for the variables that makes the query true. Prolog is not a general theorem prover, however. There are certain kinds of things it can prove and a lot of other things it cannot prove. Moreover, there are a number of places where Prolog programmers have to be aware of the specific search algorithm it uses. Fortunately, we are going to focus on very simple Prolog programs here, so we can ignore the issues of more sophisticated Prolog programming.

42.2.5 Data Structures

So far, we have said little about what kind of data objects Prolog can work with. We have seen the simple strings like `john` that do not have quotes; Prolog calls these *constants*. We have also seen numbers, and variables. Prolog also supports real strings with double-quotes. And the version of Prolog we use will be able to work with arbitrary Unity objects. More on that later.

Lists are notated with square brackets and can contain any kind of data, including variables. The built-in predicate `member` (*Element*, *List*) is true when *Element* is an element of *List*. And so the rule:

```
color_name(C):- member(C, [red, blue, green, black, white]).
```

says that C is a color name, if it is a member of the list `[red, blue, green, black, white]`, that is, if it is one of the constants `red, blue, green, black,` or `white`.

Finally, prolog supports record structures. Instances of structures are notated like constructor calls in C++ or C#, only without the keyword new. So to create a person object, you say `person` (*fieldValues*…) where *fieldValues*… are the values to fill in for the object's fields. If we wanted to break up the representation of the first and last name components of the people in our age database, we could do it by using structures:

```
age(person(john, doe), 25).
age(person(mary, shannon), 26).
age(person(kumiko, ross), 25).
…
```

Structures are matched in the obvious way during the matching (unification) process, so if we give the query `age(person(mary, LastName), 26)`, we are asking for the last names of people named Mary who are age 26, and it will reply with `LastName=shannon`.

It is important to understand the difference between structures and predicates. Each statement in your code is a statement about some predicate, either a fact or a rule. In the code above, all the statements are about the predicate `age`. Each statement tells you that it is true for specific argument values. The `person()` expressions are not referring to a predicate named `person`; they are object constructors. But that is purely because the `person()` expressions appear as arguments to a predicate, not because there is some separate type declaration saying that `age` is a predicate and `person` is a data type.

The predicate/data type distinction is determined purely by position in the statement: the outermost name in the fact is the name of the predicate, and anything inside of it is data. In fact, it is common to have data types and predicates with the same names.

42.2.6 Code as Data and Higher Order Code

The fact that code (facts, rules, and queries) and data structures all use the same notation allows us to write predicates that effectively take code as arguments. For example, the predicate `forall`, which takes two queries as arguments, checks whether the second query is true for all the solutions to the first one. For example, query

```
?- forall(person(X), mortal(X)).
```

causes `forall` to find all the `X`'s for which `person(X)`, that is all the people in the database, and for each one, test that `mortal(X)` is also true.* If all the people in the database are also listed as mortal, then the query succeeds. Another example is the all predicate, which generates a list of the values of a variable in all the solutions to a query:

```
?- all(X, age(X, 25), SolutionList).
```

finds all the 25 year olds and returns them in its third argument.

Technically, the arguments to `all` and `forall` are structures. But the fact that structures and queries look the same means that we can think of `all` and `forall` as taking queries as arguments.

42.2.7 Calling Unity Code from Prolog

We have built an open-source Prolog interpreter, UnityProlog (http://github.com/ianhorswill/UnityProlog), that runs inside Unity3D. UnityProlog allows you to access Unity `GameObjects` and types by saying `$`*name*. For example, the query:

```
is_class(X, $'GameObject')
```

is true when X is a Unity `GameObject`. The quotes around the name `GameObject` are needed to keep Prolog from thinking `GameObject` is a variable name. We can then test to see if all game objects have `Renderer` components by running:

```
forall(is_class(X, $'GameObject'),
       has_component(X, _, $'Renderer')).
```

The `has_component`(*O*, *C*, *T*) predicate is true when *C* is a component of game object *O* with type *T*. Names starting with _'s are treated as variables, so _ is commonly used to mean a variable whose value we do not care about.

* Although `forall` is built in, we can also define it by the rule: `forall(P,Q) :- \+(P, \+ Q)`, which says that Q is true for all P if there's no P for which Q is false.

Of course, there are often game objects that are not supposed to have renderers, so a more useful test would be whether all the game objects of a given kind have renderers. Since Unity is component based, we usually test whether a game object has a given kind by checking if it has a given type of component. So we might test if a game object is an NPC by checking if it has an `NPC` component:

```
npc(X):- has_component(X, _, $'NPC').
```

This says that X is an NPC if it has an `NPC` component. Now we can update our check to specifically look to see if all NPCs have renderers:

```
has_renderer(X):- has_component(X, _, $'Renderer')).
?- forall(npc(X), has_renderer(X)).
```

Finally, Prolog has a built-in predicate called `is` that is used for doing arithmetic. It takes a variable as its first argument and an expression as its second, and it computes the value of the expression and matches it to the variable. So:

```
X is Y+1
```

matches `X` to `Y`'s value plus one. We have extended this to let you call the methods of objects and access their fields. So you can effectively write ersatz C# code and run it using `is`:

```
Screen is $'Camera'.current.'WorldToScreenPoint'(p)
```

Again, note the single quotes around capitalized names that are not intended to be variables.

42.3 Writing a Static Checker

We now have everything we need to write static checkers. All we have to do is write a predicate, let us call it `problem`, which is true when there is a problem. We then give it a series of rules of the form `problem:-` *bad situation*. These literally mean "there is a problem if *bad situation* is the case." To return to our "NPCs should have renderers" example, we might say:

```
problem:- npc(X), \+ has_renderer(X).
```

Now if we give Prolog the query `?- problem`, it will automatically find all the NPCs, and check each of them for renderers. If one of them does not have a renderer, the system will reply that yes, there is a problem.

Of course, it is more useful if the system tells us what the problem is. So we will modify the predicate to return a description of the problem the rule has found:

```
problem(no_renderer(X)):- npc(X), \+ has_renderer(X).
```

which says, essentially, that "there is a no renderer problem with X, if X is an NPC with no renderer." Now if we ask Prolog if there are any problems, and if the Unity game object `fred` is an NPC with no renderer, then we should see something like this:

```
?- problem(P).
P=no_renderer($fred)
```

We can add rules for other kinds of problems too. For example, we can check that all cameras are using the correct rendering path with a rule such as:

```
problem(incorrect_render_path(Camera)):-
          Cameras is $'Camera'.allcameras,
          member(Camera, Cameras),
          Camera.renderingPath =\=
            $'RenderingPath'.'DeferredShading'.
```

(the =\= operator is a version of not equals that first runs its arguments using `is`).

Having generated a set of rules like these for finding problems, we can just add a new predicate for printing a report of all the problems:

```
problems:- forall(problem(P), writeln(P)).
```

This finds all the problems and prints them using Prolog's "write line" predicate.

42.4 Common Problems to Check

Every game has its own data structures and integrity constraints, but here is a list of common types of problems we have found useful to search for.

42.4.1 Object Configuration

One standard thing you might want to do is to look at the different game objects and check to make sure they are configured in some sensible manner. Usually, you will want to do different kinds of checks for different kinds of objects. And since an object's kind is indicated by its components, the basic pattern for these checks will look like this:

```
problem(problem description):-
     has_component(GameObject, Comp, $'Type'),
     Check for bad thing about GameObject and/or Comp.
```

For example, to check that all your particle systems are in layer number 3, you could say:

```
problem(particle_system_in_wrong_layer(GameObject)):-
     has_component(GameObject, _, $'ParticleSystem'),
     GameObject.layer =\= 3.
```

The `has_component` line finds all the particle systems and their associated game objects, and the following line checks each one as to whether its layer is 3. Any such objects that are not in layer 3 match this rule, and so are reported as problems.

Alternatively, we might check that all our objects with physics have colliders defined on them:

```
problem(game_object_has_no_colliders(GameObject)):-
     has_component(GameObject, _, $'RigidBody'),
     \+ has_component(GameObject, _, $'Collider').
```

Again, this says, essentially, "it is a problem if a game object with a rigid body does not have a collider."

42.4.2 Type Checking

Suppose your NPCs all have a `nemesis` field, which should be another NPC. Unfortunately, NPCs are game objects, and the only way to know if it is an NPC is to check to see if it has an NPC component, which the Unity editor will not do unless you write a custom editor extension. Fortunately, this sort of type problem is easy to check for:

```
nemesis(C, N):-
   has_component(C, Npc, $'NPC'), N is Npc.nemesis.
problem(nemesis_is_not_an_npc(C, N)):-
   npc(C), nemesis(C, N), \+ npc(N).
```

Here the first rule tells Prolog how to find the nemesis `N` of a given character `C`, and the second rule says that it is a problem if `C` is an NPC, and its nemesis `N` is not an NPC.

42.4.3 Uninterpreted Strings

Many games make extensive use of strings as identifiers in their data files. For example, Unity attaches a `tag` field to every game object. Tags provide a lightweight mechanism for attaching semantic information to objects, but being uninterpreted strings, the system has no mechanism for detecting typos or other problems with them. Again, we can detect these easily with a problem rule:

```
problem(invalid_tag(O, T)):-
   is_class(O, $'GameObject'),
   \+ member(T, ["tag1", "tag2", …]).
```

where `["tag1", "tag2", ...]` is the list of valid tags.

Localization is another case where uninterpreted strings can cause issues. Suppose our game uses dialog trees. Characters have a `DialogTree` component that then contains a tree of `DialogTreeNode` objects with fields for the children of the node, and the speech to display. Because of localization issues, we probably do not store the actual speech in the nodes, but instead store a label string such as "Chris professes love for Pat." The label is then looked up in a per-language localization table to get the speech to display.

But what if there is a typo in the label? To check for this, we need to compare all the speeches of all the `DialogTreeNode` objects in the system against the localization table to make sure there are not any typos:

```
problem(unlocalized_dialog(Node, Label)):-
   dialog_node(Node, Label), \+ localization(Label, _).
```

That is: there is a problem if a `Node` has a `Label` that has no `localization`. How do we find all the nodes and their labels? We find all characters and their `DialogTrees`, then walk those trees. That would be something of a pain in C#, but it is easy to do here:

```
dialog_node(Node, Label):-
   has_component(_, DT, $'DialogTree'), Root is DT.root,
   dt_descendant(Root, Node), Label is Node.label.
```

This says `Node` is a dialog node if it is a descendant of the `root` of some game object that has a `DialogTree` component. And we can define a descendant recursively as:

```
dt_descendant(N, N).
dt_descendant(Ancestor, Descendant):-
   Children is Ancestor.children, member(Child, Children),
   dt_dewwscendant(Child, Descendant).
```

The first statement says nodes are descendants of themselves. The second says that a node is a descendant of another if it is a descendant of one of its children.

Not only can we find unlocalized dialog, we can even do a reverse-check to find unused dialog in the localization table:

```
problem(unused_localization(Label)):-
   localization(Label, _), \+ dialog_node(Node, Label).
```

And we can check for the localization string by calling into whatever the appropriate C# code is:

```
localization(Label, Localized):-
   Localized is $'LocalizationTable'.'Lookup'(Label),
   Localized \= null.
```

which says that `Localized` is the localization string for `Label` if it is the nonnull result of calling `LocalizationTable.Lookup()` on the `Label`. The one problem is that this does not work for reverse lookups. So if we wanted a version that worked for reverse lookups, we could manually do a linear search of the hash table stored in the `LocalizationTable` class:

```
localization(Label, Localized):-
       HashTable is $'LocalizationTable'.stringTable,
       member(Pair, HashTable),
       Label is Pair.'Key', Localized is Pair.'Value'.
```

This works for both forward and backward searches. There are more efficient ways to code it, but for an offline static check, you may not really care if it takes an extra second to run.

42.5 Case Studies

We have used this technique in two games. The first is a research game called *MKULTRA*. *MKULTRA* uses a number of domain-specific languages for planning, dialog, etc. One of the problems with DSLs is that they often lack basic support for generating warnings

about calls to undefined functions, wrong number of arguments, etc. Using the techniques above, we were able to quickly write rules to walk the data structures of the DSLs and check them for integrity, generating error messages as needed.

The other case study is SomaSim's *Project Highrise*. A tower management game that uses its own serialization system. We found that with an afternoon's work, we were able to write a series of rules that found two pages worth of issues. Although an afternoon's work is not nothing, it is less work than would have been required to track down the issues manually. Moreover, they can be rerun each time the assets are changed, saving further work. Moreover, they can be distributed to potential modders, making it easier for them to produce user-generated content, while also reducing the tech support load on the company.

42.6 Conclusion

Modern games incorporate elaborate, ad hoc, asset databases. These databases have integrity constraints that need to be enforced but are often left unchecked due to the cost of writing custom checking code. However, declarative languages provide a cheap and easy way of implementing automated checking. Although we discussed the use of Prolog, other declarative languages, such as SQL, could also be used.

References

Bratko, I. 2012. *Prolog Programming for Artificial Intelligence*, 4th edition. Boston, MA: Addison-Wesley.

Pereira, F. and S. Shieber. 2002. *Prolog and Natural Language Analysis*, digital edition. Microtome Publishing. http://www.mtome.com/Publications/PNLA/prolog-digital.pdf