What Empirical Modelling tools should look like

0523756

Abstract

I discuss some of the problems with existing Empirical Modelling tools from the perspective of a new user coming from a programming background; most notably the claim that the system of multiple notations for different tasks is confusing. I make some proposals for an object-based definitive notation which can be extended without re-introducing the same problems.

1 Introduction

It is my opinion that the current Empirical Modelling tools, while powerful, are unintuitive for new users, especially those who are more accustomed to conventional programming.

The current system of having multiple notations (with different mappings of observable names to the same Eden namespace) is inconsistent and confusing.

However, I believe that the addition of a simple object-oriented syntax would remove the need for these different notations, which would allow new users to pick up the tools with much greater ease.

2 The problem of consistency

During my discussions with Beynon, it has been impressed upon me the importance of domain-specific capabilities; and it is obvious that the tools required to model a relational database, for instance, share very little with the tools necessary to control line drawing on a screen.

The current mechanism for accomplishing this is to develop a new notation for each problem domain: however these different notations are not only distinct in their capabilities but also in their syntax and appearance.

These differences can be small things, like whether to put a semicolon at the end of a line, or they can be larger and more important, like the Scout notation having a largely different syntax from any others. Some notations have dynamic variable typing while others do not; some define the constant PI while others use the lowercase version pi.

Furthermore, it is sometimes frustrating to integrate multiple notations, because in Eden, all observables share one global namespace: and when introducing new notations, some use this namespace directly (e.g. Scout), while some will add a prefix to all observable names (e.g. Donald).

I imagine this has been done to avoid naming clashes – where two different parts of a model wish to use the same observable name – however its effect is to make what should be a simple task (linking different parts of a model) more complicated.

The problem is even worse when using Cadence, because it is simulating an object hierarchy which internally does not in fact exist: accessing Cadence observables usefully from an-

other notation is convoluted to the point of infeasibility.

This might be acceptable if everything could be easily done from within Cadence, but since the system of having multiple notations mandates a change of notation to access additional functionality like line drawing, this extra functionality cannot be used to its full effect together with Cadence.

3 One notation for everything

My proposed alternative is to have just one consistent notation, which would be flexible and extensible enough to model different systems *without* modifying the core language.

It should be noted that all the existing notations at some level map down to Eden definitions: they simply add syntactic sugar and additional functionality at the same time. So how can we have this level of extensibility while retaining a common syntax?

I believe the answer is to have a built-in object data type. Objects are an intuitive way of grouping sets of observables that are logically connected in some way, and this is something that is sorely lacking in Eden at the moment.

The real strength of having an object data type is their flexibility. They can represent any data structure and so are well-suited to the idea of a consistent notation.

There are current prototypes to extend Eden to introduce this functionality (Cadence) and to implement dependencies on top of an object-based framework (Doste), but each of these approaches has their own advantages and disadvantages.

An object in the sense I discuss here (and as implemented in those notations) is nothing more than a mapping of key-value pairs, where the value can itself be another object. The keys may be thought of as property names, but there is in fact no reason they must be strings, or meaningful at all.

This in contrast to strongly typed languages such as Java or in C++, where given an object's type you know exactly what properties it has (and these cannot change), which is unlikely to be a good fit with the Empirical Modelling paradigm. Instead, an object should be able to have any property defined or redefined at any time, and the context in which you use an object determines the properties which are examined.

This is commonly known as duck-typing, after the so-called duck test:

"If it looks like a duck and quacks like a duck, it is probably a duck."

For example, if you consider a point to be something with an x and a y coordinate, then any object with an x and a y property (which would have to be numbers?) can be treated as a point.

It should be made clear that the introduction of objects does not replace definitive notations, but rather makes them more flexible. Even if used for nothing else, the ability to create hierarchical structures (which also mitigates against naming clashes) would make current definitive scripts more structured and more readable.

4 Operators and functions

Functions are a tremendously useful feature to have, and I believe that a notable part of my annoyance at the multiple notation system is the fact that they can only be defined in Eden and only used in a subset of notations.

The ability for the result of a function to be assigned by dependency (so that it is updated when a parameter is modified) makes them especially powerful.

Operators are even more important, because they are functions in a more intuitive and convenient form: `a+b` is essentially just `add(a,b)`.

In some languages it is possible to redefine the basic operators to behave differently in certain situations, and I believe this may be necessary to compete with the expressive power of the multiple notation system.

The ability for custom operators alone to replace specialised notations is suggested in a tiny way in my database.example file, which attempts to demonstrate how small syntax changes are all that might be required to convert an Eddi script to a hypothetical unified notation.

It is also a concept which I feel fits very well with the "what would happen if I..." approach to experimenting with a model.

However, care needs to be taken to avoid confusion over what will happen for any given calculation: if the operation you define is unintuitive, the situation may be no better than currently it is with inconsistent notations.

There are many open questions in this area, which I unfortunately have no answers to at present:

What happens if two different things try to extend the same operator?

How do you ensure that a given set of operators are consistent with expectations?

Could there be different sets of operators, of which only one would be active at a time? My instinct that this is too close to having multiple definitions, but I mention the idea for the sake of completeness.

Is it best to attempt to create a generalised algebra which will work for arbitrary objects? For example, the addition operator applied to two objects could add each attribute separately: for example `{a = 5, b = 5} + {b = 5, c = 5}` would equal `{a = 5, b = 10, c = 5}`. This would certainly give correct results for points (just an x and a y attribute), but would it in generally be intuitive, and can a full complement of these generic operations be designed? And what is the result if you attempt to add an object to an integer?

Should the user be allowed to (re)define their own operators, or should they be built-in at a higher level?

These will have to be considered and carefully prototyped. Probably it will take much experimentation to discover the best approach here.

5 A drawing hierarchy

A line can be considered a set of two points, and a point can be considered to be an x and a y coordinate.

It is tremendously intuitive to be able to consider these properties directly. Currently this is possible from Donald but less intuitive in Eden: an object-based syntax would make it

If we further suppose every drawable object can (but does not have to) contain a property "children", which itself contains other drawable objects, then these child objects could be drawn relative to their parent. Then moving or rotating the parent would also move the children.

This would allow groups of drawable objects to be transformed together: a task which I find very useful in conventional graphics programming.

This functionality has been prototyped by Nick Pope and Sam Gynn in Doste using the Warwick Game Design library, although their work focused on UI widgets rather than generalised line drawing.

A further (non-working) example can be found in my drawing.example file written in a hypothetical object-based definitive notation.

6 Implementation

It is my hope that this is not just an idealistic theory, but rather a practical solution to many of the problems that the Empirical Modelling tools

currently pose. For it to be useful, however, it must be implemented and then functionality added to bring it up to the capabilities of the current system.

There are currently two object-based definitive notations which could serve as a basis for this work.

6.1 Doste

Most of what I discuss here either exists already in Doste or could be built on top of it instead.

However, more work would have to be done in the area of extending the language to support the same diversity of models as is currently possible in Eden.

6.2 Cadence

The current implementation of Cadence is a feasible platform for these ideas. I have attempted to construct several models using it but there is a constant need to revert back to Eden to do anything. For example it is necessary to use the Eden notation to define functions, and the Donald notation to draw to the screen.

However I am informed that an implementation not based on Eden is under development, so those problems should not be relevant when that is complete.

Acknowledgements

Many thanks to Meurig Beynon for his work in shaping this paper and explaining his perspective on the various topics I discuss.

Also I would like to thank Nick Pope for his work on Doste and for introducing me to it through the Warwick Game Design library.