ODIN: A Conceptual Framework for an object extended EDEN

Joe Butler 1100422

Abstract

ODIN demonstrates how EDEN may be extended to support concepts offered by object orientation such as the association of variables and replication of objects. This article summarises the potential that the conceptual framework for ODIN offers to EDEN. An HTML5-Javascript ODIN application has been implemented to demonstrate and explore the effectiveness of the extensions proposed.

1 Introduction

Empirical Modelling [4] (EM) is an ongoing research programme at the University of Warwick, founded by Dr Meurig Beynon [1] in 1983. Unlike traditional programming methods - Empirical Modelling aims at developing frameworks for interacting with computers, that support the way that humans naturally think about the world.

EDEN is a multi-agent hybrid definitive-procedural programming language initially developed by Edward Yung [2] in 1989 to support the philosophical foundations of Empirical Modelling. EDEN has many incarnations, most recently JS-EDEN - A Javascript implementation by Timothy Monks [3] in 2011. JS-EDEN (commonly referred to as EDEN) is the current flagship of the tools constructed to support EM.

ODIN is a "weak-object" extension of EDEN. That is to say it aims to employ a select few fundamental features of the object orientation paradigm for programming to enhance EDEN - in particular the replication of objects and associated relationships between variables. It specifically does not attempt to suggest that all aspects of object orientation will benefit / be appropriate for / be applicable to EDEN / EM. It is paramount that the suggested extensions do not conflict with the underlying motivations of Empirical Modelling.

2 Current Issues

2.1 The Association of Structures

As model complexity increases, the rate of increase of the number of observables in the model grows significantly. Because the observables in EDEN are not structurally related, each observable is recorded at the top level name space. This quickly becomes an issue with any non-trivial model for a variety of reasons:

1. Selecting names for observables becomes increasingly difficult, especially so when similar structures are replicated.
2. Inspecting the observables database during construction is difficult when there are so many observables, regular expressions are the only way to bypass this issue currently.
3. Locating related observables is sometimes difficult without the assistance of a graph representing the dependencies between observables. The complexity of the connections mean that this task is often difficult for the user.

2.2 The Replication of Structures

Currently the replication of structures require a manual copy and paste of a script, in the best case with text editor replace functionality. A variety of observations emerge from this:

1. If the replication is complex it will require many code replacements and inevitably some degree of manual rewrite; this can be tedious and time consuming for the modeller. Errors are also frequently made as a result.
2. In practice structures are frequently reused and extended, as a result: subsequent changes to one copy of a script will fail to update the replication when intended. Bugs will arise as a result.

3 How Object Concepts Help

Object orientation provides a framework for the ownership of variables within objects. A 'book' object can own other variables which represent the properties associated with that book, e.g. weight, height, title, price etc. Many object oriented programming languages offer a useful notation to access these parameters: the dot notation.

The dot notation allows you to refer to the parameter of an object by succeeding the name of said object with a dot followed by the parameter's name as follows:

```
jug.leftside
book.title
person.age
```

Creating a tight relationship between these parameters offers convenient solutions to many of the issues outlined in the previous section.

3.1 The Association of Structures

The modeller will no longer frequently run out of names for observables. Two trees of observables 'leftjug' and 'rightjug' can now have the same names for all of their parameters e.g.:

```
leftjug.leftside
rightjug.leftside
```

There is no longer an observable "name explosion," sorting and categorising also becomes trivial. Instead of having 50 observables per jug at the top level name-space there is only one, all of the other observables conform to an expandable tree structure beneath the top level 'jug' observable. The tree created by this, provides a comfortable way to visually inspect structures.

3.2 The Replication of Structures

The replication of large groups of observables is now straightforward. One can simply take the root of any sub-tree of observables within the model and replicate the entire sub-tree. This can be executed with a single statement.

As parts of the sub-tree of observables will often contain references to other components within the sub-tree[1], part of that reference will be the name of the root due to the structure of the dot notation. These can easily be isolated for automatic replacement e.g.:

Consider the following two definitions within the subtree 'leftjug'.

[1]See ODIN JUGS

```
leftjug.liquid.width is leftjug.width;
leftjug.liquid.height is -leftjug.
    currentContent;
```

By replicating the entire 'leftjug' subtree with 'rightjug', all definitions and names containing 'leftjug' can be replaced with 'rightjug' and an entirely new tree of observables can be blanket created in one step:

```
rightjug.liquid.width is rightjug.width;
rightjug.liquid.height is -rightjug.
    currentContent;
```

This replacement can be made in one statement:

```
rightjug copies leftjug;
```

The user's script is reduced in size to a tiny fraction of what it would have been if all of the definitions were made individually. Updates to the left jug are also propagated to the rightjug[2] as long as they are made before any intentional explicit redefinition later in the script. It is still possible to inspect/edit a full description of the state of the model with each individual definition made seperately - This is what is represented internally.

4 Extensions Arising from the Dot

Several extensions naturally emerge from the introduction of the dot notation. They are not fundamental to the theory of extending EDEN to support "Object concepts", but a demonstration of how EDEN could benefit should it adopt the suggested framework.

In JS-EDEN, 'drawables' such as lines, rectangles, images etc... are recorded in a single observable through the use of a function e.g.

```
myLine is Line(100, 100, 200, 200, "black");
```

The properties of the line are taken as parameters to the function. A drawback of this method is that modellers unfamiliar with certain aspects of the call have to refer to documentation in order to use the function. This can be difficult when the modeller does not readily know:

1. The order of the parameters.
2. Which parameters are required and which are optional.
3. The form of the argument - (whether a list, a string of a list, a string, or an observable is required.)

[2]This operation is state dependent, just like '=.'

Considering the above definition it is possible to reference the properties of the line as values in the following way:

```
temp is myLine.x1;
```

Support has been added in demonstration implementation of ODIN for the automatic creation of the parameters of structures such as 'drawables' as observables. If one wishes to create a line in ODIN, it is efficient to make use of the natively implemented structure 'Line':

```
myLine copies Line;
```

This statement will replicate all of the observables associated with Line and assign them with default functioning values. This way the modeller can see exactly what form the values of the parameters require. It also permits many more configurations to be added to structures feasibly. The user then will not have to define a long list of parameters, only changing specific parameter observables if they require so - much like if they wished to make rightjug differ slightly from leftjug. In this case, the above definition is equivalent to making the following set of definitions:

```
myLine copies Abstract;
myLine.x1 = 0;
myLine.y1 = 0;
myLine.x2 = 100;
myLine.y2 = 100;
myLine.colour = "black";
myLine.thickness = 1;
myLine.visible = true;
```

...with the exception that abstract structures do not contain the system knowledge that Line, Circle etc do - ie: how and whether to associate sub-observables with drawing a line to the canvas.

Processes could also be implemented in ODIN using a structural framework. Parameters can be included to represent the statements that are to be executed, the observables that are to be watched, and various other configuration options that may be useful to the modeller e.g.:

```
myProc copies Process;
myProc.watches is naughtyObservables;
myProc.execute is [myProc.s1, myProc.s2];
myProc.s1 = "{errors++;}";
myProc.s2 = "{alert("error");}";
myProc.enabled = false;
```

5 The Benefits of Structural Definitions

Traditional EDEN defines '=' and 'is' as its definitive operators. The '=' operator assigns the value of the RHS at the time of assignment to the LHS (state dependent). The 'is' operator defines the LHS to be whatever the RHS evaluates to at any given time (state independent). ODIN defines two additional operators 'mimics' and 'copies' (state dependent) to accommodate definitions over structures.

5.1 New Operator 'Mimics'

The 'mimics' operator defines the LHS identically to the RHS. This allows the modeller to alter the definition of the target observables without affecting an observable which is conceived to be "doing the same thing."

```
B is A;
C mimics B;
```

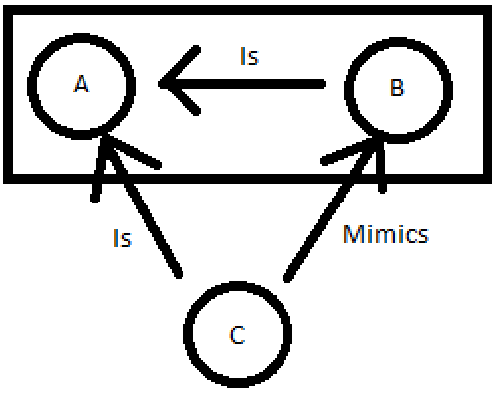


Figure 1: The above figure illustrates how the previous two definitions are equivalent.

There are two reasons why this operation is useful:

1. The modeller does not require the complete definition for A to replicate it.
2. Changes to B will not result in changes to C.

B and C are both separate observables that use A. If the modeller contemplated a replication of B without concern for how it was defined, the use of 'is' would be creating a dependency the modeller may not have anticipated.

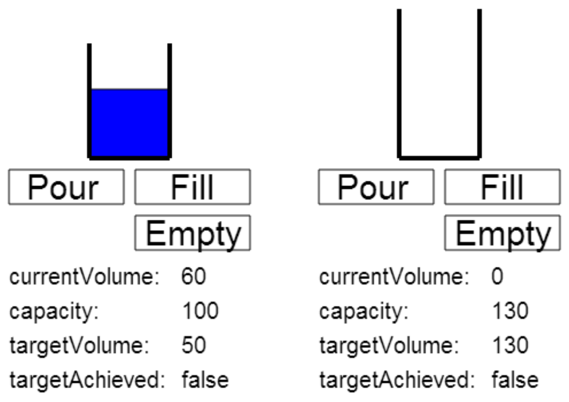
5.2 New Operator 'Copies'

When a definition is made to the root of a structure in ODIN, the sub-components of the definition within that tree are replicated and the same operator is applied to all sub-components. If the 'is' operator is used, changes to the target structure's sub-components would always result in identical changes to the respective replicated attributes, thus the motivations for 'mimics'. The 'mimics' operator replicates structures so that the copy is identical, but detached from its target. The use of 'mimics' on large complex structures however, may yield unexpected and undesirable results, as each definition in the replicated structure still contains identical definitions to that which it replicates; making the replicated structure difficult to separate from the original.

The 'copies' operator allows for a much more powerful replication - unlike 'is' or 'mimics', 'copies' will result in detached working copies of structures at every level. Specifically designed to support the replication of structures, 'copies' replicates an observable with all of its sub-components. The names and definitions are replaced with new names and definitions where occurrences of the target structure's name have been replaced with the name of the observable on the LHS of the statement.

As testament to the power of the 'copies' operator, and by extension the ODIN concepts introduced in the paper. The JUGS model has been re-engineered using the demonstration implementation of ODIN.

6 ODIN or EDEN: A Model Comparison



```
currentVolume:   60        currentVolume:   0
capacity:        100       capacity:        130
targetVolume:    50        targetVolume:    130
targetAchieved:  false     targetAchieved:  false
```

Figure 2: JUGS Model (ODIN version)

The JUGS model was originally conceived by Meurig Beynon[1] in 1988. It is a simple yet powerful model for conveying the benefits of Empirical Modelling as a paradigm for programming. JUGS has been remodelled in many major incarnations of EDEN including tkeden and JS-EDEN. It has now been implemented in ODIN. Some notable comparisons are summarised below:

- The ODIN JUGS Model comprises 76 manual statements which account for 377 single observable definitions, whereas the JS-EDEN JUGS Model comprises of approximately 55 manual statements which account for roughly 55 observable definitions.
- The ODIN JUGS Model does not include animation and its respective timing control structures as its JS-EDEN implementation does.
- The ODIN JUGS Model contains a significantly richer specification for layout than its JS-EDEN implementation.
- The entire manual input required by the modeller to produce the right jug from the left jug along with all of its respective layout properties and functionality in the ODIN JUGS Model consists of just 6 statements.
- The ODIN JUGS Model has just 2 observables at the top level name-space: 'rightjug' and 'leftjug.' whereas the JS-EDEN model has all 55.

See Appendices for full documentation of the demonstration ODIN implementation.

7 Conclusion

In EDEN all observables are hierarchically equal - and functional dependency is the only way to consider the relationship between observables. In ODIN the primary way of considering the relationships between observables is taken from object orientation. The mental relationships formed by adopting the object-like associations between observables fit well with the way humans understand relationships in the world. This would suggest the extensions outlined by ODIN are harmonious with the primary motivations of EDEN and Empirical Modelling.

As a result of adopting certain features that object orientation offers, the replication of structures becomes significantly easier for the modeller. A number of other efficiencies are also afforded.

References

[1] W.M.Beynon and R.I.Cartwright. *Empirical Modelling Principles for Cognitive Artefacts*, Proc. IEE Colloquium: *Design Systems with Users in Mind: The Role of Cognitive Artefacts*, Digest No. 95/231, 8/1-8/8, December 1995.

[2] Edward Yung, *EDEN: An Engine for Definitive Notations. MSc thesis*, Department of Computer Science, University of Warwick, UK (September 1989).

[3] Timothy Monks, *A Definitive System for the Browser. MSc dissertation*, Department of Computer Science, University of Warwick, UK (September 2011).

[4] Karl King, *Uncovering Empirical Modelling. MSc thesis*, Department of Computer Science, University of Warwick, UK (January 2007).