A definitive notation for behaviour in Empirical Modelling?

0322756

Abstract

Definitions in Empirical Modelling are used to represent state, however it is possible to use definitions to describe behaviour, as shown in this paper. To show the differences and benefits of behaviour definitions a new language called Doste is implemented based entirely on these behaviour definitions. An existing EM model will be implemented in Doste for this purpose. Current EM tools have no ideal solution for modelling agency so the aim of this investigation is to see how definitions could be used for state and behaviour. Ultimately the idea is to come up with alternative and possibly better tools for Empirical Modelling which do away with procedures and at the same time allow greater integration with the operating system. This paper is meant as an introduction to these ideas and showing the possibility for future work in this direction.

1 Introduction

A definitive notation is one in which the programmer/modeller can write a set of definitions used by a runtime system as a means of representing dependency. A spreadsheet is an example of definitions for describing state as is the use of definitions in Empirical Modelling (EM) [Russ97] . Using definitions to describe behaviour is altogether different although actually a subtle difference, where the definitions describe state change and not current state. A difference that will be clarified in this paper. Current EM tools [Ward04] do not have a clean way of describing behaviour as they have to use procedures that do not fit well with EM principles or have limited functionality. This is where using definitions for behaviour as well as state or instead of state may be beneficial and so is to be explored. To compare and analyse the two different types of dependencies a new language using only behaviour definitions was created along with a modelling environment for it. Using this language an existing empirical model called 'Jugs' [Bey89] has been implemented for the purpose of showing the different approaches and how behaviour definitions can work.

The beginning of the paper will introduce Doste and explain the use of behaviour definitions for modelling. It will then go on to explain the differences and compare to Empirical Modelling.

2 Doste

Doste is a pure definitive language [Bey85] for behaviour based on previous work [Pope06] and Empirical Modelling ideas. For this paper an interpreter and runtime system for the Doste language has been created which allows a modeller to construct models with Graphical User Interface components along with some OpenGL features for 3D models. The language is based on the mathematical function and has similarities to functional programming, although there are significant differences which will be mentioned later. It can also be thought of as a prototype-based object-oriented language instead of the more common class-based languages [ACS03] , but again there are significant differences.

The only type in Doste is an object which is a set of attributes that are name-value pairs, both of which are also objects. Each value part can either be a constant object or a definition based on other objects and their attributes. You could think of it as a kind of database with some similarities to what is described in a paper by Garrett and Foley [Garr82]. Here are a few simple examples to give an idea of the syntax:

The following constructs an empty object and puts it in attribute test of the system object.

```
system(test) = {};
```

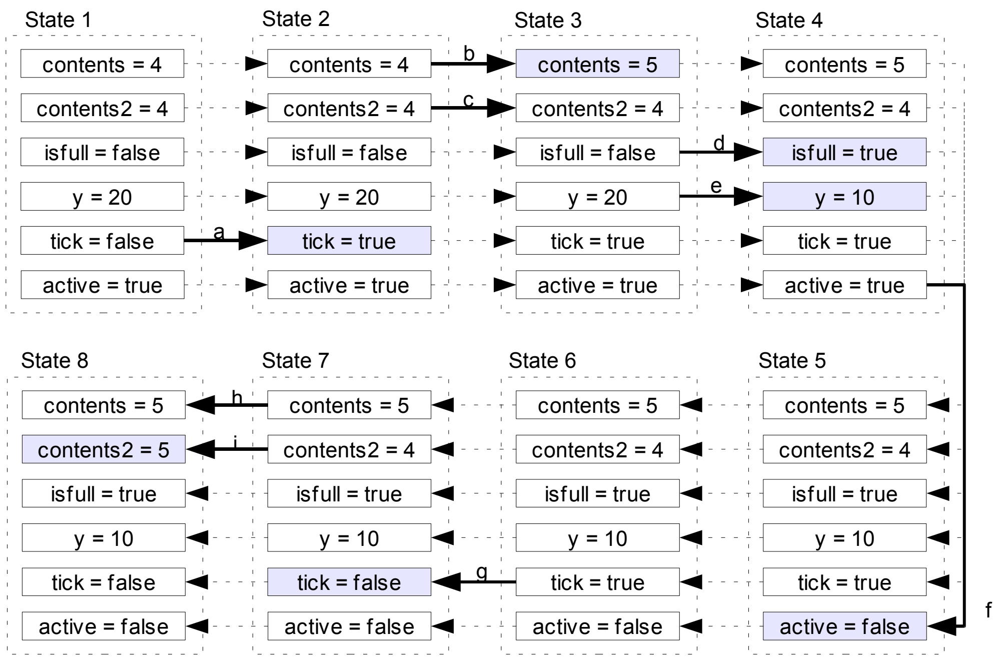

Figure 1: A state transition diagram to illustrate how definitions are evaluated for one clock cycle of the jugs model. The example given shows contents filling and becoming full at which point it stops. Note how nothing changes between state 5 and 6, this corresponds to idle time. It is also possible to see that two changes can occur at the same time, these can actually be done in parallel. Further, transitions 'c' and 'h' cause no change and so the system stops evaluating them, all the dashed lines correspond to definitions that do not change and are not calculated.

Here is an object with three attributes that can be used to represent a font.

```
this(font) =    {
                (colour, "red"),
                (size, 12),
                (bold, true)};
```

Since everything must be an object the string "red" must be converted by the interpreter into the following:

```
{(0, 'r'),
 (1, 'e'),
 (2, 'd'),
 (size, 3)};
```

Definitions are similar to having methods in an object and they do correspond well to functions in functional languages as they can have no side effects and only return an object. There is an important difference to note here, although the objects are based on mathematical functions they are not like functions of a functional language because they have to be explicitly enumerated. Definitions on the other hand are more like functions and do not have to be enumerated, they simply re-evaluate when some part of them changes. This seems to allow for a side-effect free functional type language that can store state and hence is useful for database applications and the like. Allowing these definitions to be recursive means you can not only describe state and dependency over space but over time as well, it can define state change.

Below is an example of a simple non-recursive definition that will return the **`true`** object if a jugs capacity is equal to its contents, otherwise **`false`**. Here **`current`** refers to jug A and **`this`** is also jug A, however this is of little importance and the definition can be used unchanged for any jug due to object-oriented encapsulation.

```
current(isfull) =
        this(capacity)
        (compare)
        (this(contents));
```

The following is equivalent to an if or switch statement in most languages, however the condition is on the last line not the first. Note that it is also a self-referent definition that remains the same while the clock is true but when it is false it becomes the value of contents. Such definitions stop evaluating when no change occurs otherwise it would be infinite.

```
current(contents2) =
        {
        (*, 0),
        (true, this(..)(contents2)),
        (false, this(..)(contents))
        }(system(devices)
                (clock)(tick));
```

The above definition describes the behaviour of contents2 with regard to the clock. There is a state between the clock going true and the value of contents2 being updated, as shown in the diagram in figure 1. This extra state is something that does not exist in definitions that only describe state and its inclusion allows for behaviour definitions. It may not be clear why contents2 is needed, without it the contents of the jug will fill or empty instantly, contents2 prevents it from changing by more than 1 unit for each clock tick.

3 Modelling with Doste

In order to demonstrate how Doste could be used for modelling an illustrative example will be given. This example is based on the empirical model called 'jugs' [Bey89] that was originally taken from an old educational program. A simple basic model is first constructed and then to show the dynamic modelling process the base model will be modified and extended step by step to show clearly how one would write a program or model. Some terminology has been borrowed from Empirical Modelling however there are some slight changes to interpretation that will be discussed further in the next section. A detailed account of the modelling process will not be given as it follows the same principles as Empirical Modelling.

These models can be constructed and changed at runtime in Doste, there is no need for compilation. This gives the user the ability to rapidly explore what-if questions about the model just as in a spreadsheet. Such openness is vital for modelling purposes.

The base jugs model includes two jugs, one of 5 litres and one of 7 litres, it also has buttons with which the user can fill or empty either jug. It is an incomplete jugs model because there is no way as yet of pouring the contents of one jug to the other. The first few examples show simple changes of state, the final example describes the process of adding Pour behaviour to the existing jugs using definitions.

This changes the capacity of jug A:

```
        system(jugs)(A)(capacity) = 6;
```

The statement below will redefine isfull to !(contents < capacity) which is more correct and restrictive than the original which only compared contents with capacity and so failed when contents was bigger than capacity.

```
    system(jugs)(A)(isfull) =
            this(contents)
            (less)(this(capacity))(not);
```

Now for a more complex example illustrating the adding of pouring behaviour and interface to the model. The full code for the jugs model and this extension are on-line [Doste], the code given here is only an example and is incomplete. First we need to add a button that will allow the user to pour:

```
%using
system(models)(jugs)(window);
current(Pour) = {
     (widget, button),
     (text, "Pour"),
     (enabled, this(..)(..)(canpourAB)
         (or)(this(..)(..)(canpourBA))
         (and)(this(..)(..)(pourAB)(not))
         (and)(this(..)(..)(pourBA)(not)))};
```

The important part to look at is `enabled`, this is true only when it is possible to pour and we are not already pouring something. To make this work however the observables canpour and pour need to be created. canpourAB is true if it is possible to pour from jug A to jug B. pourAB is true if we are actually pouring from jug A to jug B, note that it refers to itself so it stays the same until it cannot pour any more. The following definitions are added and duplicated accordingly for the other case.

```
system(models)(jugs)(canpourAB) =
        this(window)(EmptyA)(enabled)
        (and)
        (this(window)(FillB)(enabled));

system(models)(jugs)(pourAB) = {
        (true, true),
        (false, this(..)(pourAB)
            (and)(this(..)(canpourAB)))
        }(this(window)(Pour)(onclick)
            (and)(this(canpourAB)));
```

This still is not enough, the definition of contents for each jug now has to be slightly altered to allow its value to change if it is being poured from or to. The only changes that have been made is to make it increment if the user clicked fill or if pour is true, and to decrement if emptying or the opposite pour is true. Jug A is shown, jug B is the opposite.

```
%using system(models)(jugs)(A);
current(contents) = {
     (true, this(..)(contents2)(add)(1)),
     (*,{
         (true, this(..)(..)(contents2)
              (sub)(1)),
         (*, this(..)(..)(contents))
        }(this(..)(..)(window)
                (EmptyA)(active)
              (or)(this(..)(..)(pourAB))
              (and)(system(devices)
                (clock)(tick))))
     }(this(..)(window)(FillA)(active)
         (or)(this(..)(pourBA))
         (and)(system(devices)
              (clock)(tick)));
```

4 Comparing Definition Types

Both Doste and Empirical Modelling have the concept of observables which is the same as variables only there is a correspondence to real world or abstract quantities that have meaning in the model. Dependency is however modelled completely differently in the two systems. Definitions in Empirical Modelling are atomic and are in effect rules that agents follow when making a change, there is no possibility of being halfway through updating all the definitions. As a result it does not make sense to have recursive definitions as there is no state between the initial and final. Doste however does have states existing between evaluation updates although some do happen atomically in parallel which has similarities to what Eden does. By having extra states it is possible to use recursive definitions which just add more states before reaching the final one.

Here is a very simple example showing the difference between the two. This example does not include self-referent definitions but it is not too difficult to see how it would work.

Eden	**Doste**
`a = 5;`	`system(a) = 5;`
`b is a*a;`	`system(b) = system(a) (mul)(system(a));`
`c is a+2;`	`system(c) = system(a) (add)(2);`
`d is b+a;`	`system(d) = system(b) (add)(system(a));`

If an agent does a = 6; or system(a) = 6; then in Eden that agent also makes all the changes needed based on the definitions. In Doste it only makes that one change and the definitions themselves are responsible for evaluating to make the changes, which results in extra state transitions. The following state transition diagrams are produced for Eden and Doste respectively:

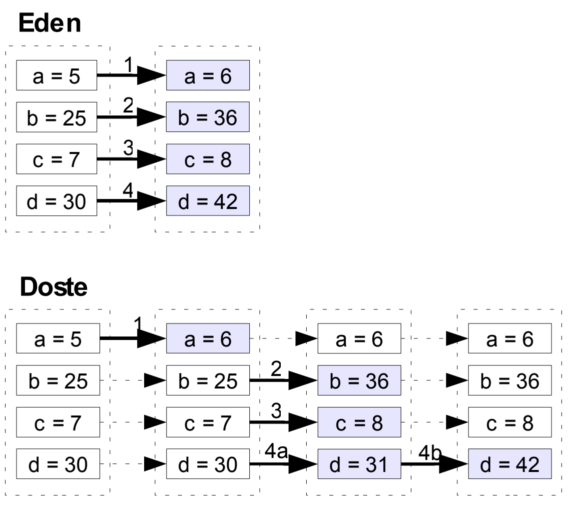

Figure 2: Shows the different state transitions of Eden and Doste. Doste has two extra states that are visible to definitions. The numbered arrows correspond to the same definitions, note that 4 becomes 4a and 4b in Doste as it is evaluated twice.

Notice how Doste has two extra visible states that can be used by definitions. Significantly, the value of d is changed twice corresponding to a partial result. Ultimately both systems get to the same state but because Doste has these extra visible states definitions can be self-referent and describe/cause a sequence of changes. It does however sacrifice some of the atomicity of definitions and creates problems if an external agent made a change during one of the partial states, although currently in Doste external events can only happen when the definition evaluation system is idle and so behaves correctly.

5 Relevance to EM

Empirical Modelling has three principles, Observable, Dependency and Agency. Observables are equivalent to variables for storing state, dependencies describe their atomic relationships and agents cause redefinitions and state change. In Eden, a tool for Empirical Modelling, procedures are used to describe the agents. Definitions can be used for everything except when a self-referring definition would be needed which in effect is to do with time or a sequence of states. Procedures have to be used in this case however the system can be abused because procedures can also be used for all other cases and definitions become redundant. It is sometimes difficult to know what should be a definitions and what should be a procedure. This issue needs to be resolved by somehow requiring the use of definitions except in the self-referent cases or by using a different type of definition in those cases.

Doste only has definitions so there is no issue of abuse, however self-referent definitions are difficult to design and so frequently go into infinite loops. This can also be a problem with recursive functions. The other issue is that dependency updates are not atomic which in a truly concurrent situation could cause certain states to be missed. The current implementation of Doste does not allow external agents to change anything until dependency evaluations become idle which means it appears to behave the same as Eden. If, for example, an agent could do a = 7; in the above example in state 2 then the value of 'd' would never become 42. Should therefore such evaluations appear atomic externally but not be atomic internally and so allowing recursive definitions?

This question is a very important one if Doste or, more generally, behaviour definitions were to be used in some way for Empirical Modelling. Presently it is possible to appear to describe state with definitions in the same way Eden does, but it also allows for self-referent definitions. There is no clear separation in Doste which is similar to abusing procedures in Eden. So why is it important to separate the two concepts? A very debatable issue that depends on your view of dependency, do dependencies exist over time or not and are real world dependencies maintained instantly with no partial transitions. In reality things are not instantaneous and things do depend on themselves so behaviour definitions make more sense, however when considering agents who are entirely outside of that model universe and can make changes at any time you have problems. For ease of understanding it may be a good idea to in some way separate the two even if ultimately they use the same system.

A benefit of behaviour definitions in combination with the object-oriented nature of Doste is that it is simpler to interact with the computer because an external device or OS signal only has to change one observable which will cause other definitions to be evaluated. In Eden interacting with the OS typically requires special functions or procedures which increases the complexity of the language and moves away from EM principles.

6 Doste Potential

In addition to exploring the modelling potential of Doste other uses have been considered. As a continuation of previous project work [Pope06] Doste can be shown to integrate well into an operating system as a replacement of the file-system but also as a kind of virtual machine in which all programs and devices are run. Preliminary analysis has also hinted at the potential for Doste to be distributed over a network and used concurrently by multiple users, something which would be very useful to Empirical Modelling. Another possible application which relates to Empirical Modelling is its use in a game engine as the scripting and database system, such a game engine in currently under development by a small group of people to test its potential in this area.

With regard to existing EM tools, there may be a possibility of converting Eden procedures into behaviour definitions, however this is proving quite complex and would have to be automated. If this could be done then Doste could be the underlying platform of Empirical Modelling in the future and would be a more well defined system.

7 Conclusion

A definitive language for behaviour is capable of being used for modelling. Its fundamental principles are Observables and Dependencies (OD) where dependencies are behavioural ones. This is different from current Empirical Modelling principles of Observable, Dependency and Agency (ODA) where dependencies are for state only and so agency is required as a separate concept. It is not possible in this paper to compare the relative merits of either modelling framework as there are not enough models written using Doste to compare with. However, for the simple Jugs model Doste seems as capable as existing EM tools are at modelling it with perhaps some advantages of not having ambiguous language constructs (procedures) and by allow encapsulation of observables into objects.

Adding behaviour definitions to EM is far more in keeping with its principles and as this paper shows it is possible that the two ideas could be brought together. The best solution at present seems to be to add a layer above Doste which separates the two types of definitions for ease of understanding but internally uses only behaviour definitions. Adding syntactic sugar or a good visual interface to Doste would also improve its ease of use.

Acknowledgements

This paper would not have been possible without the time and effort of W.M. Beynon in explaining Empirical Modelling and how Doste differs from it.

References

[ACS03] Arnstrom M. Christiansen M. and Sehlberg D. (2003). Prototype-base Programming

[Bey85] Beynon W. M. (1985). Definitive notations for interaction. Proc. HCI'85, ed Johnson and Cook, Cambridge University Press, 23-24, 1985.

[Bey89] Beynon W. M. et al. (1989). Software Construction Using Definitions: an Illustrative Example. CS-RR-147.

[Doste] Pope N. (2007). Doste Models. [on-line]. (url: http://www.dcs.warwick.ac.uk/~csudek). Accessed 25/1/2007.

[Garr82] Garrett M. and Foley J. (1982). Graphics Programming Using a Database System with Dependency Declarations. ACM Transactions on Graphics, Vol. 1, No. 2, April 1982, 109-128.

[Pope06] Pope N. (2006). Prototype-Based Object-Oriented File System. 3rd Year Project, Department of Computer Science, University of Warwick.

[Russ97] Russ S. B. (1997). Empirical modelling: the computer as a modelling medium. BCS Computer Bulletin, Volume 39, Number 2, April 1997, 20-22.

[Ward04] Ward A. (2004). Interaction with Meaningful State: Implementing Dependency on Digital Computers.