Introduction to definitive notations and tkeden

Empirical Modelling Research Group
http://www.dcs.warwick.ac.uk/modelling/

1 The concepts

Real world phenomena can be modelled on a computer by describing them as problems that can be solved using computer programs. These problems and methods of solving them need to be somehow represented in the programming language being used. Conventional languages (i.e. procedural languages) use variables and procedures to model the problem and have a set flow of control for solving it. The variables refer to storage locations that hold data about the objects being modelled and their values collectively represent the current state of the problem. Procedures describe what to do with the data stored in the variables and consist of sequences of instructions whose side effects change the state of the problem. The computer has to be told explicitly when to change certain variables and how to compute them. This puts added burden onto the programmer since he/she has to decide when it is appropriate to update certain variables and has to include instructions to do so in their program.

Imagine a model that contains a table and a lamp, which can be moved around by specifying their positions. To represent this you would need a program that contained, among others, variables holding the positions of the table and the lamp. To make the lamp sit at the centre of the table, the position of the lamp needs to be the current table position, plus an offset to the centre of the table. If you wanted to move the table to a new position then the position of the lamp also needs to be changed requiring a procedure to change the table and lamp positions. Now whenever the table is moved, the lamp is moved with it. If you also want to move the lamp to any position on the table, the procedure needs to be changed to take into account the position of the lamp on the table. In addition, to get an up-to-date picture of this model, a redraw procedure needs to be called every time something is moved. All these dependencies need to be programmed in explicitly, which can become a complex task with many dependent objects or dependencies that are more complicated.

The above example shows that while procedural languages can be (and are) used for modelling, they demand extra effort that could be better spent on the actual modelling itself. It would be more convenient to be able to provide abstract definitions of objects in term of other objects being modelled. The computer can then handle these definitions implicitly, making sure that they are always true. This is especially important during development of the model, when objects and the dependencies between them need to be frequently changed. Only the actual definitions need to be changed whereas, with procedural languages, entire sections of code may need to be altered, which takes time and is prone to errors. A system that can manage definitions between objects is called a definition-based or definitive system.

Definitive systems consist of variables (referred to as observables), which hold data representing different objects, and definitions, which describe the dependencies between objects. A definition usually defines a single target variable by an expression containing one or more source variables. The target is then said to depend on the source objects and the expression governs how the target should be computed from them. These definitions are handled by the system, which ensures that whenever a variable is changed, the variables that depend on it are re-evaluated to reflect this change.

The following illustrates a typical definition:

```
t = f( s1, s2, ..., sn );
```

where t is the target variable, s1...sn are the source variables and f is an expression involving the source variables. This definition states that t depends on s1...sn and is computed from these source variables using the expression f. Should any of the source variables change then the expression is recomputed automatically so that this definition will always be true. An example of definitive principles in use is the spreadsheet, which consists of a grid of cells (observables) which can hold values or formulae (definitions). The formulae state how the value of a particular cell can be computed from other cells automatically. Whenever any cells are changed, formulae referring to that cell are recomputed and a display action is invoked by the system. In the case of the spreadsheet, the display depends on all the cells and the display action is an implicit action since it is pre-defined by the system.

Some advantages of using definitive notations are as follows:

- There is no need to remember which variables to update when a change has been made to the state of a model. This is especially useful when the environment is continuously changing.
- All variables will contain the most up-to-date value.
- They allow easy interaction for a user since response to a user's actions is immediate.
- They provide a natural way of describing the relationships between the objects being modelled.
- Most notations are run-time interpreted, so there are no long compilation delays.
- If they are interpreted then all objects and definitions can be modified at run-time, allowing a wide range of interaction and run-time changes when developing new models.

2 A general definitive notation: Eden

Eden is a general-purpose language that supports the concept of definitions. Its name comes from the name of its original interpreter, EDEN - an Evaluator of DEfinitive Notations. The language is not a purely definitive one but a hybrid language combining definitive and procedural programming. Each statement in a typical program is either a procedural style statement or a definition. When a procedural statement is encountered by the interpreter it is executed and will have the effect of evaluating an expression, assigning a value to a variable or calling a procedure. On the other hand, when a definition is encountered, an equivalent definition is set up internally, which will then be evaluated by the interpreter whenever this is necessary.

The syntax of Eden is similar to C. Like procedural languages, it has variables that represent storage locations, and can be assigned values from expressions that can contain other variables, operators and functions. Unlike C, variables do not need to be declared before being used and are allocated storage when they first appear in the program. The type of a variable does not need to be given and depends on the type of the value assigned to it. The following are types that are essentially the same as those in C:

- Integer e.g. `123` (decimal), `0456` (octal), `0x1f` (hexadecimal)
- Character e.g. `'a'`
- Floating point e.g. `1.23`, `.23`, `1.23e-15`

There are also some additional types:

- Undefined. This is represented by `@`. A variable that has not yet been defined will have this value.

- String. A string can be defined as a sequence of characters surrounded by double quotes, e.g. `"this is a string"`.
- List. The only structured type in Eden. It represents a list of data values and each element can be of a different type. An example of a list is `[ 100, 'a', "string", [1,2,3] ]`. If `l` is a variable holding a list, then `l[i]` is the `i`th element of the list and is of the same type as that element. The length of the list is given by `l#`.

Functions in Eden are similar to C, except that parameters are specified inside the function body. All local variables used in the function also need to be declared at the beginning of the function body. This is done by providing a list of them after an `auto` keyword, although their types do not need to be given. Function definitions are indicated by the keyword `func` or `proc` (generally, `func` should be used for functions that return something, and `proc` for those that do not), for example:

```
func max {
   para m;                        /* m is alias of first argument */
   auto i;                        /* local variable */
   for (i = 2; i <= $#; i++)      /* for each argument */
      if ($[i] > m) m = $[i];     /* keep max */
   return m;                      /* return max of all arguments */
};
```

Eden has the usual arithmetic, relational and logical operators for handling the numerical types (e.g. `+`, `-`, `<`, `>`, `=`, `&&`, `||`). Values can be assigned to expressions through assignment statements which are of the form `var = expression;`. Other statements in the language, used for control flow, are the usual C `if-else`, `while`, `for` and `switch`.

The language also has two additional statements for defining dependencies and triggered actions. These statements are not executed directly, but when encountered, set up the definitions internally. These statements specify that a variable, or action, depends on a set of source variables. Whenever a variable has a value assigned to it, any definition that includes it as a source variable is re-evaluated. A dependency is similar in format to a variable assignment statement except it provides an abstract definition of the variable. The syntax is the same as that of an assignment with the `=` symbol replaced by the keyword `is`. A typical formula definition is given as follows:

```
t is f( s1, s2, ..., sn );
```

where `t` is the target formula variable, `s1...sn` are the source variables and `f` is a formula which may be an expression or a function. This statement forms a definition, which says that the `t` depends on the source variables `s1...sn`. The value of `t` is re-evaluated if any of the source variables change. Hence, the difference between dependencies and assignments is that dependency definitions can be considered to always be true (target variable is equal to the expression) but an assignment statement can only be assumed to be true at the point directly after its execution. As an example, if you wanted to define the following: "C lies at the mid point of line AB, where A and B are defined independently", then this could be given as the following definition: `C is (A + B)/2`.

The second definition statement that Eden provides is for triggered actions. These provide a way of specifying explicit actions, which are called when certain observables change. Triggered actions are procedures, which depend on a set of source variables. Their syntax is similar to that of a function but with a list of variables, called the dependency list, which the function depends on. This is a comma-separated list, preceded by a colon, between the function name and its body. An example of a triggered action:

```
proc print : a, b, c {
   writeln(a, " ", b, " ", c);
};
```

In this action definition, the procedure is triggered by the variables `a`, `b` or `c`. If any of these variables changes then the procedure is called, and the values of the three variables are printed out. Triggered actions are useful when some procedural side effect is required.

The final statement, and most useful when modelling, is the query statement. To find out the definition (and current value) of the observable `x`, use: `?x;`

3 A definitive notation for drawing: DoNaLD

The name DoNaLD stands for Definitive Notation for Line Drawing and this notation allows the definition of two-dimensional shapes. The definitive nature of this line drawing notation allows the points, lines and shapes to be described in terms of other points, lines and shapes. Through dependency maintenance, changes to one observable will propagate change to all the dependent observables in the model. With this graphical output, the results of modelling a problem can be seen immediately and in a convenient form. The notation itself consists only of definitions of shapes, and it is therefore a pure definitive notation. This means that there are no directly executable or procedural statements, and every statement is a definition. A script in this notation is therefore like a specification of the dependencies between different shapes and their elements.

4 A definitive notation for screen layout: SCOUT

This definitive notation for SCreen layOUT provides a notation for displaying windows and creating graphical user interfaces. It is used for describing the geometry and layout of windows on a display screen using dependency. Since it defines the layout of the screen, it provides an interface between the output of other definitive notations and the actual display. For instance, a window can display the image from a DoNaLD line drawing. The notation also provides various means of detecting interaction events (e.g. pressing a mouse or keyboard button) and entering text. This allows the development of simple graphical interfaces with each window handling its own interaction.

5 Other notations

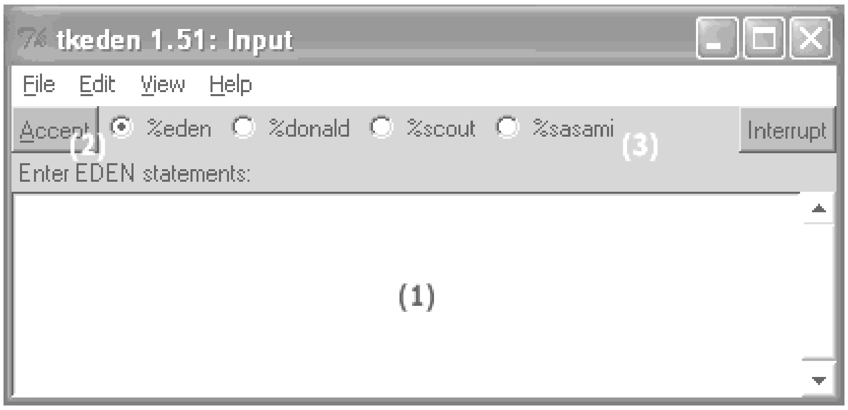
Various other definitive (and non-definitive) notations exist to provide support for more domain-specific modelling: Sasami for 3D graphics, Eddi for relational databases, LSD for distributed agent-based interaction, and many more.

For syntax and more information on Eden, DoNaLD, SCOUT and other definitive notations, please see the documentation at: http://www.dcs.warwick.ac.uk/modelling/

6 Getting started with tkeden

The primary tool for building dependency-based models is tkeden. This tool incorporates the main definitive notations (Eden, SCOUT and DoNaLD) as well as some other domain specific notations. Tkeden is currently available for Windows, Linux/Unix and Mac OS X. Binaries and sources are available from the EM website mentioned above.

To start, run the tkeden executable and an environment for interaction will be displayed. The key elements of the interface are: the text entry box (1), the accept button (2) and the notation switches (3). Statements are entered into the text entry box (1) and then executed by clicking on the accept button (2). The environment enables the modeller to switch between notations on-the-fly by selecting



the notation switches (3).

Other useful features in the environment include the ability to view all the observables and definitions currently in the model ('View → View Eden Definitions'). Basic guides to each of the main notations are available from the 'Help' menu.

Note: When running tkeden under Windows, a console window will be opened to view any text output from the environment.

7 Loading existing models

A model can be developed interactively as described above, or existing models can be loaded into the environment. To load an existing script, use 'File → Open' and select the file to be opened in the environment. To execute the loaded script, simply click the accept button.

A large variety of models are available from the Empirical Modelling Archive:
`http://empublic.dcs.warwick.ac.uk/`

When you download a model and extract the files, you will find various file types. Generally, models are often split into scripts containing different notations: .e (or .eden) refers to Eden scripts, .d refers to DoNaLD scripts and .s refers to SCOUT scripts. Models will usually contain a file named run.e (or run.eden) that can be loaded to start the model.

A model can be started by opening the 'run.e' file from within the tkeden environment using 'File → Open/Execute'. Otherwise it can be started from the command-line:

```
tkeden run.e
```

This will cause a new tkeden environment to be opened and the file run.e to be loaded. Once a model has been loaded, the tkeden environment can be used to interact with the definitions and enter new statements in any of the notations.

8 Revision history

This document was derived from Andy MacDonald's *Notes on Definitive Notations* with material added by Antony Harfield on introducting tkeden.