Chapter 4

Cadence: A Prototype Tool

Having identified some key issues with EM and EDEN in chapter 3, and found possible solutions, the next step is to introduce a new prototype tool that has been developed to go some way towards finding out whether these solutions work in practice. The prototype tool has been called *Cadence* and constitutes a major contribution to this thesis. It is one attempt at developing the dynamical semi-structured OD-net described previously. Cadence is envisioned as a future alternative to EDEN which resolves many of its issues and better enables Empirical Modelling concepts and principles to support plastic applications, as well as generally showing how EM can be useful for software as a whole. This chapter will introduce Cadence and identify ways in which to implement a semi-structured OD-net and how to make it dynamical. The implementation architecture is then given, which includes discussion of user interfaces and other extensions developed with a C++ API. Subsequent chapters will illustrate the use of Cadence with example models and explore conceptual aspects, before finally relating it all back to Empirical Modelling and concluding with how it helps support plastic applications.

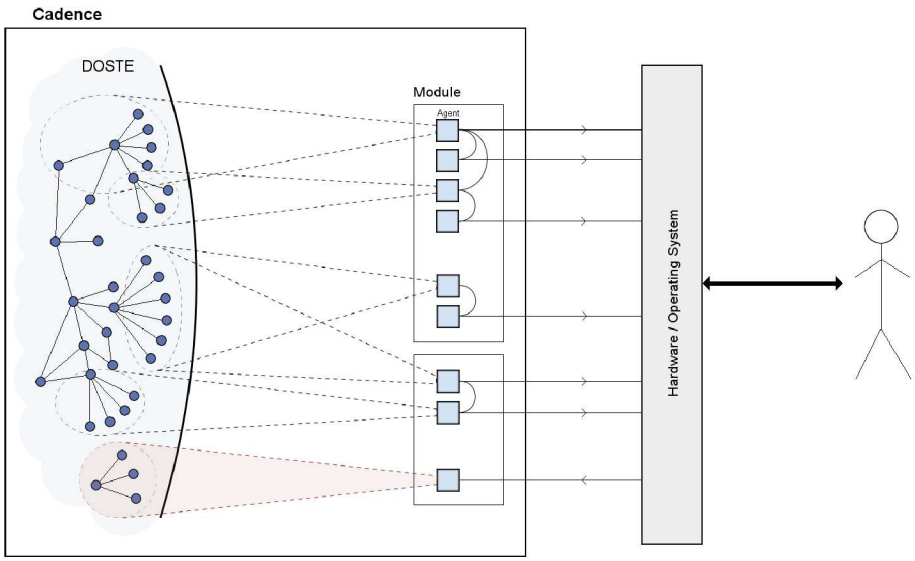

Figure 4.1: Cadence conceptual model

4.1 What is Cadence?

The name Cadence has been chosen to emphasise a creative flow of software development, not dissimilar to the creation of music or poetry, and because of its dynamical nature. Cadence is a software environment that supports the development of construals and a refinement of those construals towards programs by embodying the principles of end-user development (cf. §2.1.1) and Empirical Modelling (cf. §2.2.2). It can be thought of as an operating system[1], as a virtual machine, as a modelling tool; it is all three of these, the distinction between them being somewhat inconsequential. In other words, it is intended to be a plastic software environment.

The tool implements the idea of a dynamical semi-structured observable-dependency network, as described in §3.4, along with the modular glue interpretation for agents. There are, therefore, two key parts to Cadence which are shown in figure 4.1: the OD-net core called *DOSTE*[2] and the agents which interact with it. Agents are provided in a

[1] A version of Cadence did boot as an independent operating system on 32-bit and 64-bit multi-core systems. This version was not maintained.

[2] Definitive Object State Transition Engine, but this is for historical reasons [Pope, 2007].

modular fashion[3] and may interact with or observe the OD-net. This interaction is live, immediate and unrestricted so that construals at the personal, provisional end of the spectrum are supported. Agents may act independently or, more often, act as mediators between the OD-net and any human users via a computer's hardware and peripherals, as depicted in figure 4.1. Agents provide means of observation by interpreting the OD-net in specific ways, and also provide means of interaction through controlled modifications to the OD-net. Refinement may take place by placing more restricted agents between the user and the artefact. The focus of this thesis is on the Cadence OD-net and not so much on the agency currently supported.

4.2 Semi-structuring the OD-net

The problems identified in §3.3.1, and to some extent those of §3.3.3, show a need to group observables into either name spaces or structures of some kind, albeit flexible ones. A solution was proposed in §3.4.1 which suggested that the OD-net should be semi-structured in a graph-like manner and that there should not be classes but that sub-graphs could be cloned instead to make new structures. The core of Cadence, DOSTE, implements the OD-net as an edge-labelled directed graph in its simplest form: a *magma* (cf. §3.4.1). This section describes how Cadence has achieved this semi-structuring.

4.2.1 How to Introduce Structure

Since DOSTE is to implement structure as a graph, graph terminology will be used. However, it is helpful to relate graph structures to other concepts such as observables and objects. Figure 4.2 shows different terminology for simple graph structures. A node corresponds to some entity, object, *observational context* or value, depending on choice of terminology and the current interpretation being given to a node, and is identified by an Object-IDentifier (OID). An edge from a node represents some named property, attribute or component of it. The edges are also identified (labelled) with OIDs. Some

[3] Agents are currently C++ classes but this is far from the desired way of implementing agents.

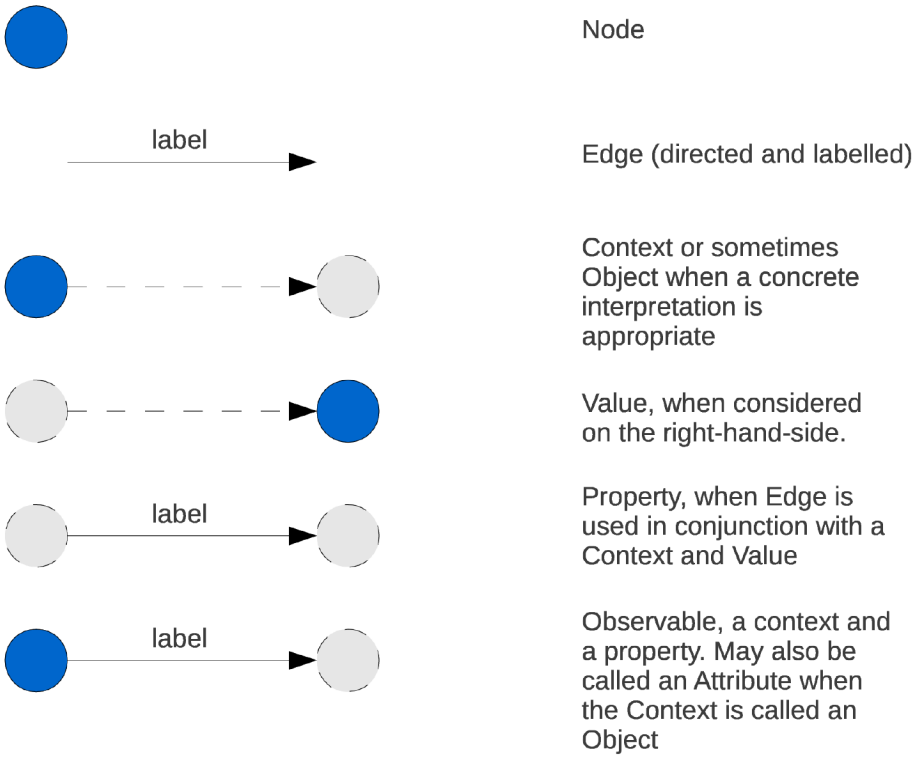

Figure 4.2: Cadence terminology for graph structures. Dark (blue) nodes and solid lines are what the terms refer to, whilst the grey and dashed lines set the context for use of those terms.

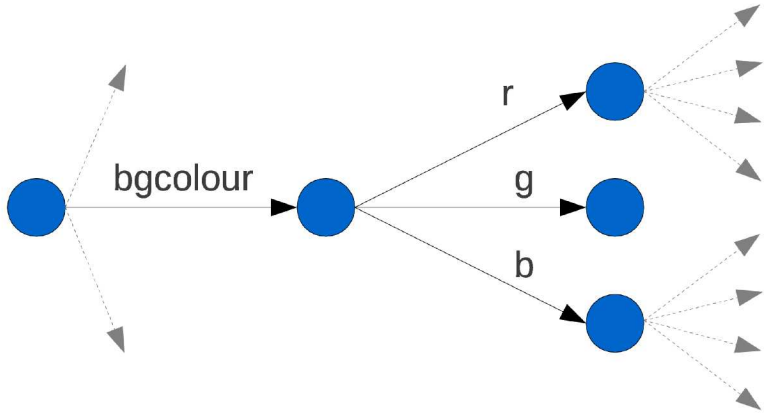

Figure 4.3: A graph example of representing a colour

OIDs can be mapped, by agent interpretation, to strings and numbers so that edges appear to have names, the same can be true for nodes. Due to edges and nodes both being identified by OIDs, it is possible to use a node OID to identify or select an edge - an important characteristic to be revisited (cf. §4.3.1). An observable is a node-edge pair which is connected to, or points to, another node that would be the *value* of that observable. The node part of an observable may be considered as a form of *observational context*, with an observable's value node either being interpreted as a plain value or an object. An observational context is typically the root focus of the current observation[4] and structures within this (i.e. the values of observables) are usually referred to as objects. Edges of *observational contexts* are observables whilst edges from value/object nodes are often referred to as attributes.

There is a need for some nodes to be interpreted in a special way to allow for numbers, but conceptually a number node is no different from any other. The graph in figure 4.3 shows an example of an observable called 'bgcolour', inside an observational context, which has three attributes for each colour component. The values of each attribute are nodes which correspond to some integer. Another interpretation, when the observational context is shifted to the colour object, is that 'r', 'g' and 'b' are observables

[4]The notion of "current observation" is fluid so what is deemed an object in one moment may be considered as an observational context in the next.

and not attributes. This shows how terminology shifts as the observation changes focus.

4.2.2 Developing a Textual Notation

A basic textual notation has been developed as a part of Cadence to enable a user to interact with the underlying DOSTE graph and as a way of describing the graph in text such as this thesis. The notation is called DASM which is short for DOSTE Assembly since it is best thought of as a form of assembly language with the potential for higher-level notations, including visual ones, to be placed above it. Originally it was compiled into a form of byte-code to be loaded by the operating system version of Cadence, but is now directly interpreted. At present it remains the only notation with which to interact with Cadence other than directly interfacing via C++ or one of the interface extensions discussed in §4.4.5. As a consequence all models and examples are based upon DASM so a good understanding is important to fully appreciate the work of subsequent chapters.

The DASM notation is unlike other semi-structured data approaches such as XML. Instead of representing static data (as a tree in XML's case), DASM represents state which is intended to be changed interactively. Individual DASM statements can be entered into Cadence (via some interface) to build up the artefact inside DOSTE[5], so in this way DASM should be considered as an interactive interface rather than a script. There is some similarity to Subtext, only DASM is a textual representation of what Subtext is trying to do more directly. Both are attempting to allow live manipulation of a software artefact's internal representation.

One of the simplest statements in DASM is a query that navigates the graph and returns the node that is reached. This involves specifying a start node and then giving the edges to follow. Nodes and edges are all labelled and these labels can be single words or numbers but may also be more abstract identifiers. There is a special reserved word called '**this**' which is the root node in the current context[6]. '**this**' may

[5]There is some similarity between DOSTE and the JavaScript Document Object Model (DOM), which XML in the form of HTML helps to build.

[6]*current context* can be changed to focus on another area of the graph and is a property of the interface through which the DASM script is being entered. It is possible to have unconnected graphs

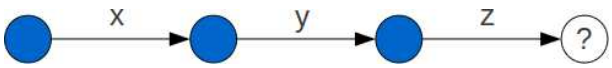

Figure 4.4: Navigating before an assignment

also be written as a *dot*. The example in listing 4.1 (cf. figure 4.4) starts from the '**this**' node and navigates along the edges 'x' then 'y' then 'z' to reach an unknown node. If the system has never been told what node should be pointed to by these edges then it will always return the '**null**' node where every edge that leaves the '**null**' node points back to the '**null**' node[7].

```
this x y z
```

Listing 4.1: DASM graph query

Following from this is a simple form of definition, shown in listing 4.2, where the node an edge points to can be changed to some fixed node (cf. figure 4.5). Instead of returning an unknown node a new node is given and the node the last edge originates from is returned. Having the origin node returned allows several assignments to be chained together, which can be seen in listing 4.3.

```
this x y z = 50
```

Listing 4.2: DASM edge assignment

The first example in listing 4.1 would now return the node '50' instead of '**null**'. It is important at this stage to think of '50' as being a representation of a node and not a numeral that represents a number. It is simply being used as an identifier and for it to be interpreted as representing a number requires other structures to be in place for numerical operations. These structures and relationships will be introduced later but for

in DOSTE with several different users using different interfaces with different contexts so that they can work independently.

[7] In this way there is never a problem with an edge being undefined.

```
this x y
    z = 50
    a = 4
    b = 5
```

Listing 4.3: DASM chained assignments

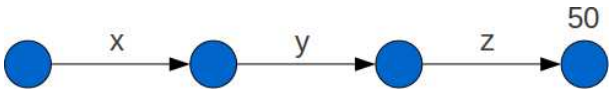

Figure 4.5: Navigation after an assignment

the moment assume the system has no knowledge of them.

To make an edge point to a node from another part of the graph, a path can be given on the right-hand-side (rhs) of the equals (cf. listing 4.4). It is critically important, however, that this be surrounded by brackets as otherwise the first identifier will be interpreted as the node the edge should point to instead of navigating the whole path. The remainder of the path would then act as a query to be combined with the left-hand-side (lhs) and will return a result, as demonstrated in listing 4.6 which shows how the example in listing 4.5 would be interpreted.

```
this x y z = (this z y x)
```

Listing 4.4: DASM assignment using a path

With what has been shown here it is possible to build simple structures and make basic queries by giving a path through the graph. Another useful feature worth introducing here is that of *context variables*. These are place holders for storing the result of some query for use elsewhere in the script which saves having to perform that query everywhere[8]. All context variables begin with the '@' symbol as shown in listing

[8]Context variables substitute in their current value when used so if the variable is subsequently changed any scripts that used it previously are not changed.

```
this x y z = this z y x
```

Listing 4.5: Incorrect assignment in DASM

```
this x y z = this;
this x y z y x
```

Listing 4.6: Interpreting an incorrect DASM assignment

4.7.

```
@xy = (this x y);
@xy z = 50
```

Listing 4.7: DASM context variables

The example in listing 4.7 is equivalent to the first assignment example in listing 4.2 but now every occurrence of '**this** x y' can be replaced by '@xy'. These context variables are an alternative to using the current context '**this**'. The current context could change, and often does, so context variables provide a stable means of referring to certain nodes.

When constructing a graph it is also necessary to refer to new nodes for edges to point to that can then themselves be filled with edges[9]. For this purpose there exists a '**new**' keyword which returns a unique and unused node. It can be placed at the beginning of any statement as a start node for graph navigation (cf. listing 4.8). If used on the right-hand-side of an equals it must always be contained within brackets otherwise it will be interpreted as a *label* (OID) rather than a keyword.

Both of the examples in listings 4.8 and 4.9 give the same result, although the second is shorter and more common in examples. With these constructs it is now possible

[9]Conceptually all nodes already exist, as do all edges. Unused nodes have all their edges pointing to the 'null' node.

```
this button = (new);
this button
    x = 10
    y = 10
    caption = "Button";
```

Listing 4.8: Node construction approach 1

```
this button = (new
    x = 10
    y = 10
    caption = "Button"
);
```

Listing 4.9: Node construction approach 2

to develop any graph structure supported by DOSTE. So far the notation shows the richer nature of observables and observational contexts than those found in the existing EDEN tool.

4.2.3 Cloning Sub-graphs

One of the benefits of having structured observables and contexts is the ability to manipulate these structures, a key problem with EDEN with regards to scalability. Cloning of structures is an example of this and is a good way of replicating a large or complex collection of observables, including their definitions. There is a large body of research and practical examples of cloning which can include some quite complex inheritance mechanisms, with Self providing inspiration for this work (cf. §2.1.3), along with Subtext and JavaScript. In Cadence cloning is a direct and complete copy with no in-built notion of inheritance. The DASM notation provides a special keyword called '**union**' that performs a copy of the object on the rhs into the object on the lhs. If there are two

edges with the same label then the one in the object on the right will replace the one on the left. The operation will return the left hand object.

```
.test1 = (new
    a = 0
    b = 4
);
.test2 = (new union (.test1)
    a = 3
    c = 6
);
```

Listing 4.10: Cloning sub-graphs

The example in listing 4.10 first creates a node with edges 'a' and 'b'. The second part then makes a node which first copies all the edges in the first and then modifies and adds to them. In the 'test2' node there are 3 edges: 'a', 'b' and 'c' with values 3, 4 and 6 respectively. Another example, in listing 4.11, shows the use of multiple '**union**' operations to create a moveable window from existing prototypes.

```
this mywindow = (new
    union (@prototypes window)
    union (@prototypes dragable)
    title = "Test Window"
    width = 200
    height = 100
)
```

Listing 4.11: Combining graphs with union

As the system is a graph, and may potentially be cyclic, there is a problem with the cloning mechanism. By default the '**union**' operation performs only a shallow clone where edges are copied but still point to the same value object. In some cases it is

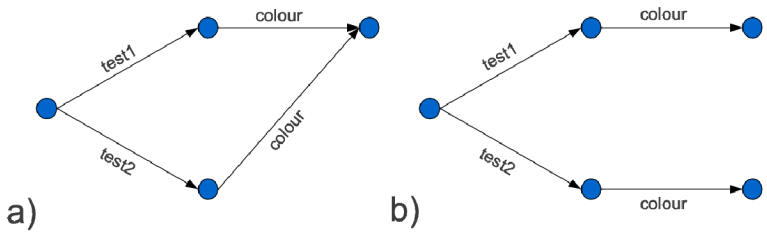

Figure 4.6: Results of shallow (a) and deep (b) cloning.

desirable to perform a deeper clone that will also clone a value node as well and use this as the value for the copied edge. DASM provides an annotation called '%deep' which flags a particular edge as needing a deep clone when the '**union**' operation is performed on it. Included in listing 4.12 is an example that would require deep cloning and the subsequent example in listing 4.13 shows how this can be achieved with the annotation. In the first case, if the colour values are changed they would change in both 'test1' and 'test2' due to the colour object being the same. The second example fixes this problem with the resulting graph shown in figure 4.6.

```
.test1 = (new
    colour = (new r=0.0 g=1.0 b=0.0)
);
.test2 = (new union (.test1));
```

Listing 4.12: Shallow clone example

```
.test1 = (new
    %deep colour = (new r=0.0 g=1.0 b=0.0)
);
.test2 = (new union (.test1));
```

Listing 4.13: Deep clone example

4.3 Making the OD-net Dynamic and Dynamical

In order for the DOSTE graph to be an OD-net it must support the current EM concept of dependency (cf. §2.2.2). Dependency allows the observable graph structure discussed previously to update coherently in the presence of external change; it makes the OD-net dynamic. In §3.3.2 limitations were identified with only using dependency in a passive sense and relying on agency for all change. A solution was proposed in §3.4.2 to make the OD-net support dynamical systems by allowing dependencies to exist across time. The proposal is to make the OD-net not only dynamic, in that it can be changed, but also dynamical, in that it changes by itself[10]. This section looks to expand upon the previous by first introducing the idea of *computation-by-navigation* and then showing how passive and active dependency can be added to DOSTE.

4.3.1 Computation by Navigation

Observables in the graph can be accessed by navigation from some starting node (the observational context) and following any number of edges until a resulting node is found. In the object-oriented tradition, nodes, when interpreted as objects, can contain edges that represent operations upon that object. The node an operation edge points to represents a *curried* function and all edges coming from that node correspond to the second parameter being given. Following one of these edges leads to either a resulting value or another function if the function has more than two parameters.

A good example is that of integers where a node represents a number and has edges for addition, subtraction and so on. An example is given in the graph in figure 4.7 which shows an addition operation for the integers 0, 1 and 2. Following the addition edge moves to an intermediate node with edges for all other integers that you may add to the first. All kinds of operation can, in principle at least, be represented by this graph navigation approach but this does require some nodes to be very large or infinite

[10] This use of dynamic and dynamical may be controversial. Dynamical is used for systems that change with time whereas dynamic has a broader meaning often associated with action (of agents).

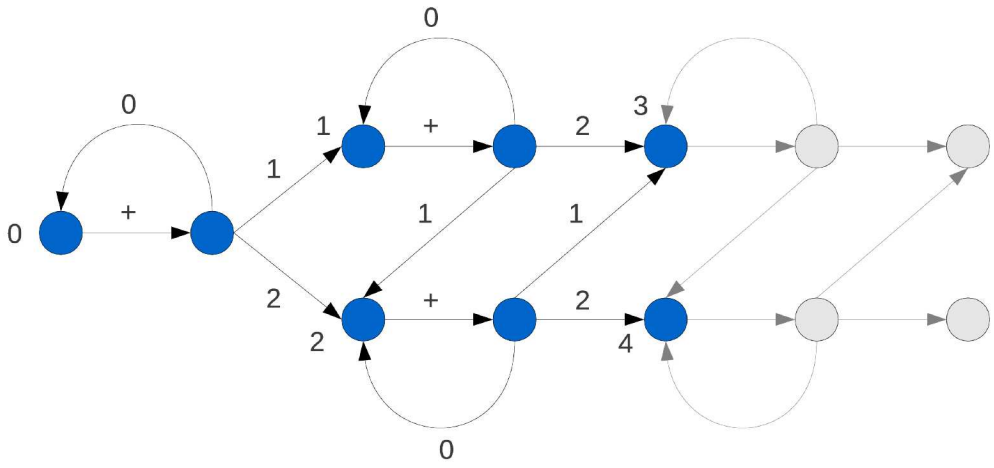

Figure 4.7: A portion of the graph for integer addition.

in extent (i.e. an infinite number of edges) and that there exist a very large or infinite number of nodes. The navigation approach also relies heavily on the fact that node OIDs can identify edges, as will become clear. What this approach to computation results in is an ability to explore logic, arithmetic and functions in the exact same way as exploring the data. There is no difference between data and code, computation becomes a form of observation as well (cf. chapter 6).

The example graph depicted in figure 4.7 can be expressed explicitly using DASM and is shown in the example script in listing 4.14. In practice a full definition of integer addition and other operators is extremely large and is considered infinite, so such a description cannot be given explicitly and must be built-in as a virtual part of the graph[11]. Conceptually, however, it works as shown.

If the definition in listing 4.14 were to be completed for all numbers and operators then arithmetic expressions can be specified as paths to follow through the graph. This takes advantage of the cyclicity of the OD-net graph.

Any arithmetic can now be performed by graph navigation (cf. listings 4.15 and 4.16). The agent that does the following is doing the computation and that agent may

[11] An alternative is to use generic definitions described later as a way of describing potentially infinite graphs in a lazy fashion.

```
0 + = (new);
0 + 0 = 0;
0 + 1 = 1;
0 + 2 = 2;
1 + = (new);
1 + 0 = 1;
1 + 1 = 2;
1 + 2 = 3;
2 + = (new);
2 + 0 = 2;
2 + 1 = 3;
2 + 2 = 4
```

Listing 4.14: DASM definition of addition

```
1 + 2 + 3 + 4 + 5
```

Listing 4.15: Simple arithmetic in DASM

```
5 * (2 + 6) / (7 * 8)
```

Listing 4.16: Arithmetic in DASM

well be the human user, not just an automated process. In a sense it is a little like a structured lookup table that a user or machine can follow to perform some computation, although not restricted to numbers. The graph describes what can be computed by observation, but not what is computed as that is entirely at the discretion of the observer. The example in listing 4.16 shows how sub-queries can be used. Sub-queries work by taking advantage of the fact that edges are labelled with the same OIDs as nodes. So the result of a query is a node that is then used to identify an edge from another node.

Much more will be said about the theory behind computation by navigation in chapter 6. The next example, in listing 4.17, defines the boolean operators. Boolean operators can be described in exactly the same way and because they are small and finite they can easily be given explicitly so are not a built-in virtual part of the graph. In listing 4.17 is the boolean 'and' operation given in DASM:

```
true and = (new);
true and true = true;
true and false = false;
false and = (new);
false and true = false;
false and false = false;
```

Listing 4.17: DASM boolean 'and' operator definition

Listing 4.18 is an example of using the 'and' operator for a boolean expression. It is easy to visualise the computation process by following the graph in figure 4.8 to evaluate the expression in listing 4.18. The node obtained after all edges have been followed is the result of evaluating this expression, in this case '**false**'.

```
true and true and false and true
```

Listing 4.18: DASM boolean logic

Conditional statements can also be expressed using graph queries and will be

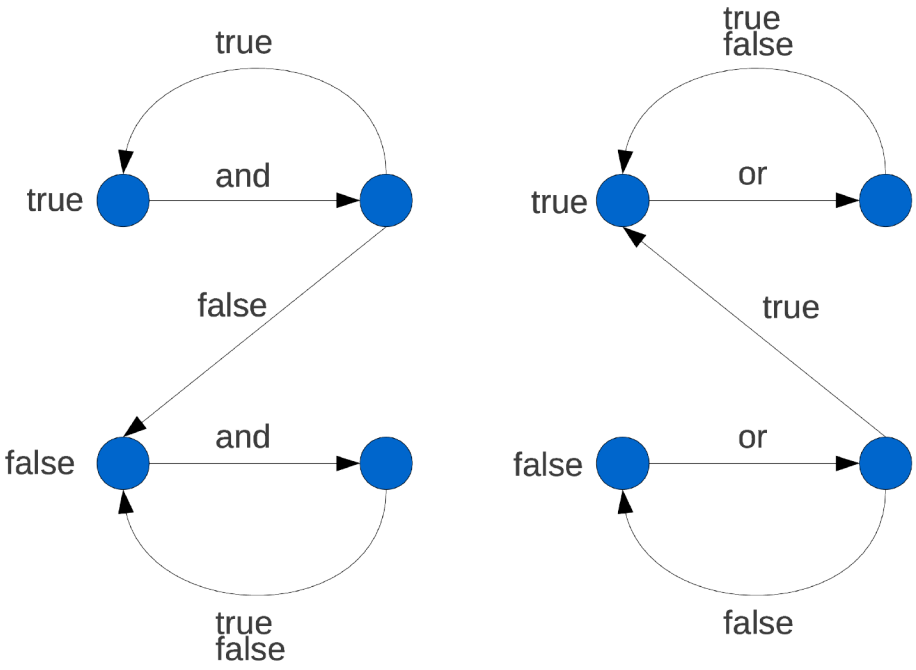

Figure 4.8: Complete graphs showing boolean 'and' and 'or' operators in DOSTE

shown in the next section.

4.3.2 Definitions for Passive Dependency

To support Empirical Modelling's original notion of dependency (cf. §2.2.2), called *passive* dependency here, edges in the DOSTE graph can be given definitions instead of directly pointing to another node. A definition, when evaluated, gives the node that the edge should be pointing to. Since computation is by graph navigation, definitions are described as paths through the graph which are to be followed to "calculate" a resulting node. Whenever any of the edges in the path followed change, either by their own definitions or by agent action, then the edge's definition is marked as out-of-date so when next observed the definition is re-evaluated. This is dependency maintenance in action.

Latent definitions, as opposed to dynamical definitions (cf. §4.3.3), describe passive dependencies and can be given in DASM using a special '**is**' keyword instead

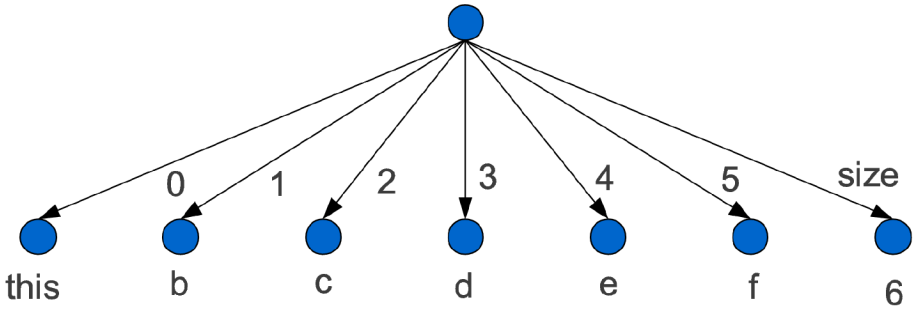

Figure 4.9: DOSTE definition represented as a graph structure.

of '=' as used previously. The simplest form of latent definition is one that acts as a short-cut, as shown in listing 4.19.

```
.a is { .b c d e f }
```

Listing 4.19: Shortcut definition in DASM

So now if any of the edges 'b', 'c', 'd', 'e' or 'f' are changed then the edge 'a' is also updated. Note how the right-hand-side of the 'is' is surrounded by curly braces instead of plain braces (contrast with listing 4.4), which indicates that the rhs should be encoded as a definition instead of immediately evaluated as a path to follow. Definitions in DOSTE are encoded as nodes and edges, so for example the definition in listing 4.19 becomes the graph depicted in figure 4.9. It is possible in DASM to assign a definition encoding to an observable using '=' with curly braces on the rhs, allowing a definition graph to be explored in the same way as any other structure. Similarly a definition structure could be built manually and then used with 'is' by giving a path (or node) on the rhs instead of a curly brace definition[12].

A slightly more complex example of a definition would involve performing some calculation. In the example in listing 4.20 a definition is used to always maintain an observable as the square of another.

[12] The curly brace syntax for constructing definitions is syntactic sugar.

```
.a = 5;
.b is { .a * (.a) }
```

Listing 4.20: Definition to square a number

In listing 4.20 '`b`' would have the value of 25 and if '`a`' were then changed to 6 it would automatically and indivisibly become 36. These kinds of definition are clearly similar to spreadsheet formula. Unlike spreadsheets they return nodes that describe structures and so is closer to, but less restricted than, Forms/3 which supports richer types (cf. §2.1.3). With a passive dependency it is not possible to observe a state where the definitions have not been updated (indivisibility of update). Internally DOSTE operates in a lazy fashion, so definitions are only actually re-evaluated upon use when out-of-date rather than as soon as they become out-of-date. This lazy approach saves a great deal of processing and is the same technique that is used in EDEN. In example 4.20 the part after the multiplication has been put inside brackets which states that this part must be done as a sub-query. Without the brackets the definition would read from left to right as a list of edges so would attempt to multiply the dot and apply '`a`' to the result. Operator precedence is unlike that in most languages, it is always left to right. This is because of the graph navigation interpretation.

There are two additional things to be aware of with regards to latent definitions and the use of passive dependency:

1. They cannot be cyclic, so must not refer to their own observable on the rhs either directly or indirectly. The example in listing 4.21 shows indirect cyclicity and will fail with an error message in Cadence[13].

2. Performing an assignment does not override the definition and so assignments are completely ignored for any edges with latent definitions.

[13] Cyclicity errors will not cause Cadence to crash, it gives a message and returns *null* as the result of the definition.

```
.a is { .c }
.b is { .a }
.c is { .b }
```

Listing 4.21: Indirect cyclicity example

Definitions may also need to use conditionals to decide what paths to follow. There are no special operators for conditional statements in DOSTE definitions as they are not required. It is possible to construct graph structures using latent definitions and node OIDs to select edges which produce the same effect as a conditional statement. A conditional is used to select an edge in a node and the value that edge points to is the result of the conditional. An example is given in listing 4.22 of such a structure.

```
.ifdemo = (new
    . = (.)
    true is { ..b }
    false is { ..c }
);
.d is { .ifdemo (.a == 1) }
```

Listing 4.22: If-object construct in DASM

In the example the edge `d` is defined to be the value of edge `b` when `a` is 1, otherwise it is the value of edge `c`. Having to construct such structures for every conditional statement would take time and so a syntactic sugar has been added to DASM that enables a more traditional way or writing such *if* statements (cf. listing 4.23). It should be noted that internally all this does is automatically generate the structure shown in listing 4.22.

Notice how, in listing 4.23, inside the true and false parts double dot is used instead of just a single dot. This is a direct consequence of the actual translation of this

```
.d is {
    if (.a == 1) {
        ..b
    } else {
        ..c
    }
}
```

Listing 4.23: Syntactic sugar for conditionals

statement because a single dot refers to the *if* object itself which then has a parent[14] set to be the original context, hence needing double dot to access the original context inside the *if*. Nested ifs are possible, as is *else-if*, and in all of these double dot is required. It always remains only double dot rather than triple because all nested *ifs* still have their parent set to the original context rather than their parent *if*. If-constructs are an example of a Cadence design pattern; there are many more such patterns that have been identified.

4.3.3 Definitions for Active Dependency

Dynamical definitions implement the active form of dependency for describing processes[15]. Instead of referring to the present they describe what the future value of an edge will be. Dynamical definitions can refer to the current value of the observable which they are defining and so allows future values to be defined in terms of the current value. Such definitions can be used as counters and for animation as well as other dynamical systems applications. In DASM a dynamical definition is given using '`:=`' instead of '**is**' but is otherwise syntactically identical. The example in listing 4.24 will

[14] Objects may have an edge that points to its parent object, and usually this is an edge labelled with a *dot*. It may not always be possible to have a single parent, but when cloning is used to generate the object the parent edge is automatically set.

[15] Dynamical definitions were at one time the only type of definition in Cadence until *hiaton* issues where identified that required the latent form of definitions. Both are therefore needed. Hiaton issues are synchronisation problems due to different lengths of dependency chains.

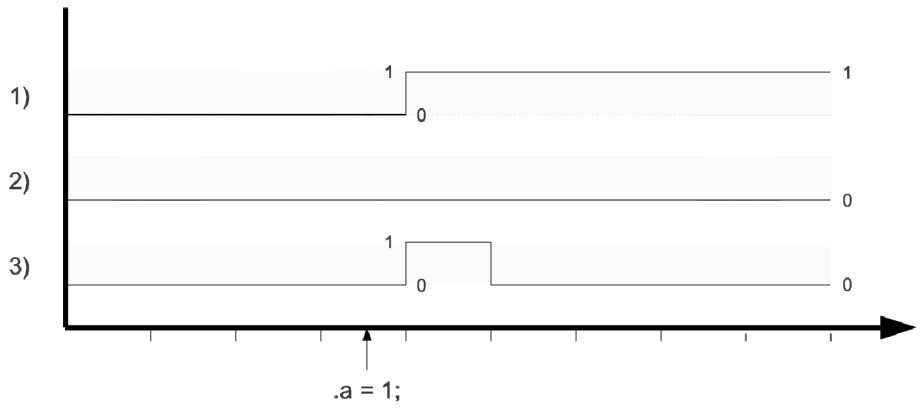

Figure 4.10: Definition comparison over time. Relates to listing 4.25.

count as fast as the machine can by having an edge add one to itself continually. Notice how 'a' must also be initialised to some initial value using assignment. With dynamical definitions it is possible to have a definition and to use assignment at the same time, unlike latent definitions.

```
.a = 0;
.a := { .a + 1 };
```

Listing 4.24: Counting with dynamical definitions

The semantics of the different kinds of definition can be illustrated using the set of possible definitions for 'a' given in listing 4.25 and the graph in figure 4.10 showing how the observable 'a' is affected over time by an assignment for each of the three possible definitions it may have.

```
1 .a = 0
2 .a is {0}
3 .a := {0}
```

Listing 4.25: Semantics of definition types

The first line of listing 4.25 shows assignment which defines 'a' to be 0 now but without saying anything about how it might change. The second line gives a constant latent definition which says that 'a' is 0 now and will always be maintained as 0 regardless of any assignments performed on it. The final line states that 'a' will become 0 in the very next instant but that right now it could be something else. If an assignment is performed when the definition on line 3 is in place then it will take effect but in the very next instant the value will revert back to 0, as is shown in figure 4.10. By default all edges can be thought of as having the dynamical definition in listing 4.26 which gets overridden by the user.

```
.a := {.a}
```

Listing 4.26: Default definition

What the definition in listing 4.26 says is that 'a' will always become whatever it already is and because a dynamical definition is used it is possible to use '=' to change its value. Removing a definition from an observable is not possible, instead the above definition would need to be reinstated to provide the default functionality. Internally definitions are only evaluated if they need to be due to dependency maintenance and as a consequence the definition given in listing 4.26 effectively becomes latent in that no change is occurring until agent intervention.

4.3.4 Generic Definitions

To a limited extent it is possible to describe generic definitions in DASM using the '$' token which gets substituted with the edge label. An experimental feature, it has already shown potential in allowing functional style node descriptions since it can act as a function parameter to be specified as needed. Previously there was an example showing a definition being used to define the square of some number (cf. listing 4.20). An alternative way of describing this operation would be to use a generic definition.

```
.square = (new
    $ is { $ * ($) }
);
```

Listing 4.27: Generic definition to square a number

```
.square 5
```

Listing 4.28: Using the generic square

Example 4.27 describes a generic definition which multiplies an edge node with itself to produce its square. This does not require the edge to be a number so can be applied to any node that has a multiplication edge. In the second example, shown in listing 4.28, the number 5 edge is applied to square and so the result of that expression will be the node 25. The definition is not instantiated until required for a specific edge, at which point the new edge is automatically constructed and the definition evaluated and cached. In other words generic definitions are lazy. '$' will match any edge that does not already exist in the object so if an edge is explicitly given it will override the generic definition. With this mechanism it is possible to provide a base case to an otherwise recursive generic definition. A recursive example is given in listing 4.29 that calculates factorial.

```
.factorial = (new
    0 = 1
    1 = 1
    $ is { $ * (. ($ - 1)) }
);
```

Listing 4.29: Factorial using generic definitions

Generic definitions and their potential are explored further in chapter 6 and is

also discussed as future work. It would be one way of supporting abstraction but is difficult to directly experience through exploration of the graph as it has no concrete instantiation.

4.4 Implementing Cadence

Cadence has been written in C++ and is supported on Windows, Linux and MacOS, now available as an open-source project on github [Pope, 2011]. It may also be, and has been, compiled as an independent operating system[16]. To fully appreciate how Cadence operates an understanding of its evaluation mechanisms and internal representations is useful, especially for dynamical definitions. This section will briefly discuss the architecture of Cadence and will introduce the C++ API used for developing extensions. A few specific extensions are then given, including a 2D development interface, a 3D graphical interface for games and support for network distribution as well as integration with EDEN.

4.4.1 Architecture Overview

The architecture of Cadence is reasonably simple and is depicted in the diagram in figure 4.1. Of real interest, however, is the architecture of the DOSTE core of Cadence and it is this which is to be discussed here. The DOSTE architecture is shown in figure 4.11. Primarily the architecture is about the routing and processing of events in precise ways to achieve the desired result of indivisibility of latent definitions and the correct "behaviour" for dynamical definitions whilst still allowing for agent interaction and observation. All interaction, observation and internal communication is done using events. Agents generate events to modify and observe the system, these are then sent to one of four queues to be routed, at the correct time, to an event handler. Handlers may also then generate events to maintain dependencies. It is the handlers that store

[16] Not available on GitHub.

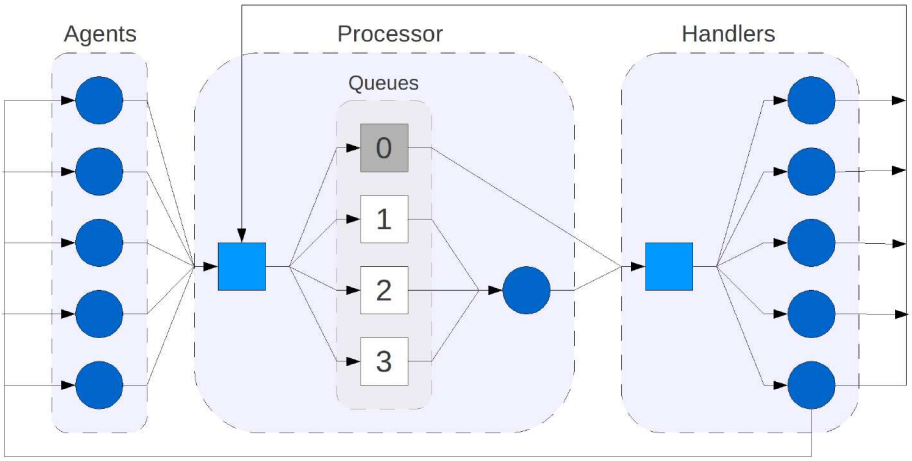

Figure 4.11: DOSTE architecture diagram showing the core components.

some representation of the graph and edge dependencies, with some handlers being used for virtual parts of the graph.

The choice of this particular architecture has come out of considerable experimentation and many different attempts. The nature of the events, the fact that there are four queues for different event types, what types of event there are and how handlers and agents operate is all very precisely controlled and orchestrated to give a reliable representation of an OD-net that can be used for Empirical Modelling. This removes the huge burden of orchestration from the modeller, a problem that Edwards was looking to overcome with Coherence (cf. §2.1.3). There were two additional motivations behind choosing this particular architecture:

1. Network distribution by allowing events to be transparently routed to handlers on different machines.

2. Concurrency by allowing multiple processors to work on processing queued events without causing conflicts[17].

[17] Although not currently maintained, at one time Cadence did support the use of multi-core processors for concurrent processing of events and this worked reasonably efficiently and was reliable. Problems arose when trying to link this with traditional technologies such as OpenGL which did not work well with such fine-grained concurrency.

What is not shown in figure 4.11 is how agents and handlers can be added by loading dynamically linked libraries at any time. Due to this, the core DOSTE part of Cadence is relatively small and consists primarily of the processor part and a framework for adding agents and handlers.

4.4.2 Events and Queues

Event-programming is common in operating systems because it is the most effective way of dealing with asynchronous concurrent communication. It is especially useful for user interfaces where actions can occur at any time in any part of the program but also for inter-process communication (IPC). To think of each definition and agent in DOSTE as a miniature process shows how fine-grained and extensive the potential concurrency is and how important effective IPC is. It is for this reason that events have been chosen since actually modelling DOSTE as a collection of active processes is exceptionally inefficient[18]. Instead DOSTE must be entirely event-driven.

Events are represented in DOSTE as small packets of information that have a standard structure. These structures are passed through the system to be processed at the correct time in the correct place either synchronously or asynchronously. Since the event mechanism is mostly asynchronous it is necessary to have at least one event queue which stores the events before they get processed. In practice however, more than one queue is required because different types of event need to be processed at different times. There are two reasons for this:

1. by grouping all read-only events and write-only events together you can remove the need for some of the concurrency locks and slightly improve performance.

2. more importantly though there is a need to synchronise and get ordering correct in order for the correct behaviour to be observed when dealing with dynamical definitions.

[18]The major operating systems (Windows and Linux) do rely on processes polling for events, but this is acceptable given the comparatively small number of processes involved.

Type	Queue	Description
GET	1	Return the node an edge from a node points to.
GETKEYS	1	Return a list of all edges from a node.
SET	2	Change the node an edge from a node points to.
DEFINE	2	Change the definition of an edge from a node.
NOTIFY	3	Notify an edge that its definition is out-of-date.
ADDDEP	4	Record a dependency on a particular edge.

Table 4.1: Main DOSTE event types

If the system fails to properly order the events it can end up with incorrect and almost random results or no result because some definition fails to get triggered. An example might be if an *add-dependency* event was processed after another *set* event, in which case some definition might not get *notified* of the change that occurred and incorrect values are the result.

Events have the following structure:

type The type of event determines how it is processed

destination The node to which this event is being sent

parameters(n) A number of parameters (max 4), each of which is an OID

result An optional result OID for some events

All events are some action to be performed to a particular node and therefore events have a destination node. It is this destination that determines where the event is sent for processing by a handler. Different nodes get managed by different handlers and this is determined by the OID.

The type attribute determines what action to perform on the destination node. Table 4.1 shows some of the more important types of event. When they are sent they are added to one of three queues depending on the type of event: *write* (SET and

DEFINE), *notify* or *dependency*. Each CPU will go through one queue at a time and process each event. Read events (GET) are always synchronous (don't get added to a queue) but should only happen during the notify queue cycle. Write events get added to the first queue and when processed they may cause notification events to be generated if there are other observables with dependencies on the one being changed. Notify events are added to the next queue which is processed after all the write events have been processed. A notify event may cause a definition to be evaluated which can generate read events and add-dependency events. The read events are performed immediately so that the result of the definition can be calculated. Finally, a notify event will generate a single write event to actually perform the change, but this gets added to the first queue and so will not be processed immediately. This is important because it means all other definitions that still need to be processed can use the old values, otherwise the results would be non-deterministic. Add-dependency events get added to the next queue after notify events so that they are all performed before any write events. If this was not the case then some writes would occur before the dependencies are added and so some definitions will not be correctly notified of a change that does affect them. Once a cycle through the queues is complete the whole process starts again with the first write queue. Each cycle is called an instant[19].

The above description is for dynamical definitions, there is a slight difference for latent definitions because a notification of being out-of-date needs to happen immediately when a change occurs. Latent definitions will evaluate only on-use-when-out-of-date as opposed to dynamical definitions which will evaluate each instant when out-of-date.

A simple example of a definition evaluation can illustrate the flow of events. The example in listing 4.30 describes a definition that centres a button's x-coordinate based upon its width and the width of the window it is in.

As soon as the button x definition is entered by an agent, the agent generates a

[19] An instant is one discrete time step of the dynamical system.

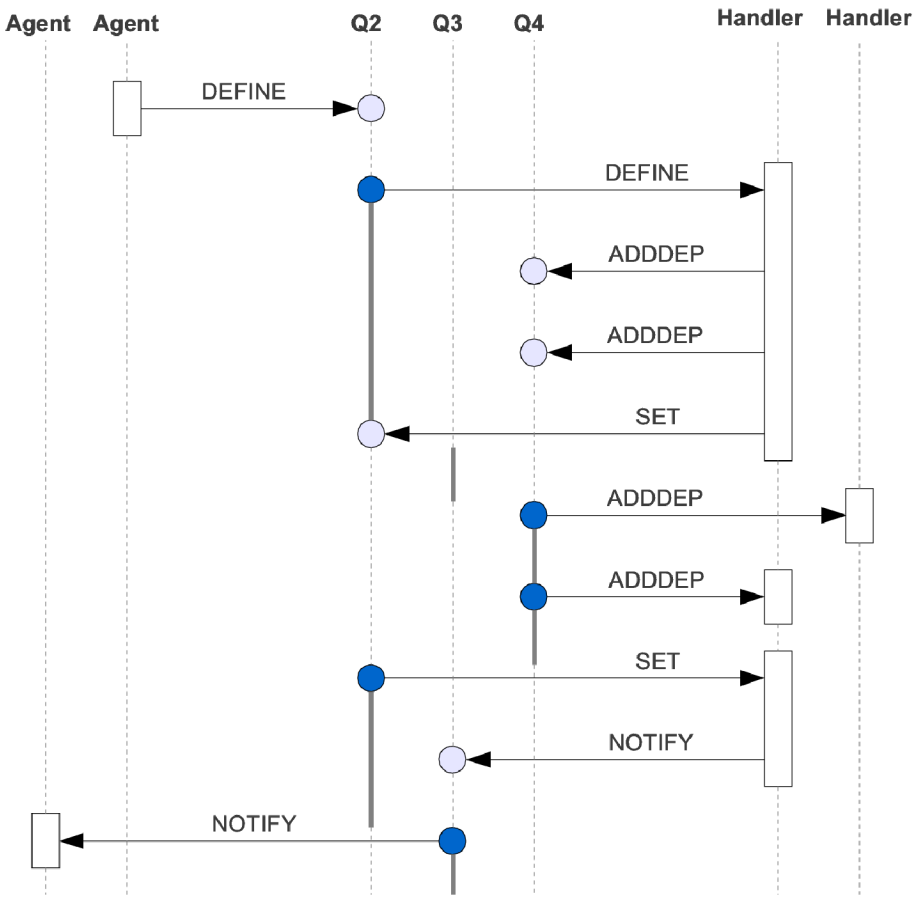

Figure 4.12: DOSTE Event Flow Example

```
.button x := { @window width / 2 - (.width / 2) };
```

Listing 4.30: Button centering example

DEFINE event[20]. Figure 4.12 shows the sequence of events generated by this example, although read events have been left out for clarity. The left most handler in the diagram corresponds to the button node. The other handler is for the window[21]. As can be seen, each queue is processed in order and is highlighted to show when it is being processed. The final consequence is that a NOTIFY event gets sent to another agent that represents a graphical interface being told to redraw the button.

4.4.3 Handlers and Agents

A handler receives events for a particular set of nodes for processing. A handler must internally have some representation of the nodes and edges it is responsible for, as well as keeping track of definitions and dependencies. Each handler gets registered with the event router when it is installed so that the router knows to send events for particular nodes to it.

Handlers may be virtual in that they do not actually store information but generate it on-the-fly or send it over a network for processing by a remote handler. Some handlers may connect directly to hardware. The handlers available in Cadence by default are:

Local Stores nodes and edges in memory.

Network Forwards events to another machine.

Numbers Simulates a virtual graph for numbers.

IO Maps edges to IO ports on the machine.

Files Maps file system into the graph.

Agents Manages agents and forwards events to them.

Custom handlers can be added at any time to extend the system. The last handler listed above is of interest, it represents all agency. Although the diagram in

[20] Although the agent first constructs an object to represent the definition that was entered.
[21] In practice both of these handlers are likely to be the same.

figure 4.11 shows agents and handlers as being separate, this is not actually the case. Agents are managed by and contained within an agent handler. This has been done so that agents can take advantage of the existing event mechanisms to receive notifications of changes that trigger them.

4.4.4 C++ API

The C++ API constructed for Cadence will be briefly discussed since this is a key part of its flexibility. The main focus for the API was to make the addition of new agents as easy as possible so that new functionality could be added by anyone with basic C++ knowledge. To fit with the always running interactive nature of Cadence it is also possible to add these new agents at any time whilst the environment is running.

C++ comes with a large number of features for customising the language. This includes operator overriding, templates and macros that are used extensively in the API. An agent in Cadence is an object in C++ which can respond to changes that occur in the graph. It does this by having event handlers which get called when specific observables change. An agent is then able to observe and change the graph using a simple mechanism as well as use any other C++ libraries to, for example, draw graphics on the screen.

Each agent in C++ is an instance of a class that describes that type of agent (cf. listing 4.31). DOSTE provides a run-time type system which enables all these agents to be automatically constructed as required. To achieve this each agent class (or agent type) is registered with DOSTE and given a label so that Cadence can look at a graph node and determine what type it should be[22]. When another agent requests an agent object from a particular node it will look at the type attribute and find the appropriate class to use to make an instance of that object. In this way the programmer does not need to know the class to use as the system will determine this at run-time. In addition, if an instance for that object already exists then that is returned instead of

[22] Ideally this would be done as duck-typing but is currently done by checking an explicitly given label identifying its type.

```
class Assigner : public Agent {
    public:
    Assigner(const OID &o): Agent(o) { registerEvents(); }
    ~Assigner() {};
    OBJECT(Agent, Assigner);

    BEGIN_EVENTS(Agent);
    EVENT(evt_condition, (*this)("condition"));
    END_EVENTS;
};

OnEvent(Assigner, evt_condition) {
}
IMPLEMENT_EVENTS(Assigner, Agent);
```

Listing 4.31: C++ agent example

a new object being created. With this the programmer does not even need to worry about constructing or deleting agent objects. A benefit of this approach of automatic run-time typing and construction is that dependencies between modules can be removed since one module does not need to be aware of another in that it does not need to know about specific classes in other modules. It is an example of dependency-injection where the OD-net describes what C++ objects to construct.

4.4.5 Graphical Interfaces

Using the provided C++ API, two different graphical interfaces have been developed for Cadence as modules by providing a collection of agents. Neither of these are a focus of this work and are there to enable visualisation and interaction but have not been developed to be especially user friendly. EUD and HCI research would be needed to find a more direct and friendly way of manipulating the Cadence OD-net[23].

[23] Self, Forms/3, Subtext and many others provide inspiration for how this may be done.

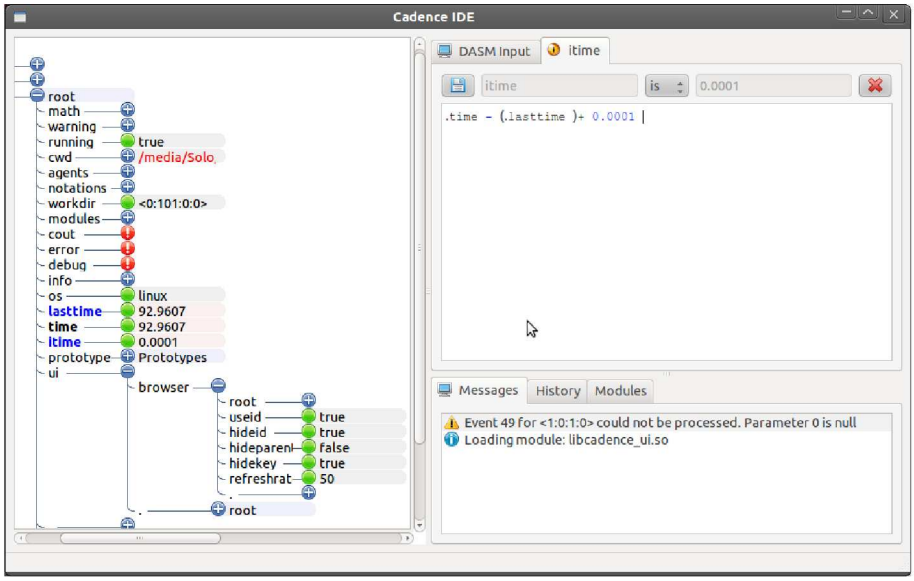

Figure 4.13: Cadence User Interface Module

The first is a development interface, as shown in figure 4.13, which shows a representation of the OD-net as a tree structure and allows for the input of DASM statements to manipulate the graph. The visual representation of the graph can also be manipulated in basic ways to make changes. The OD-net visualisation is live and will be immediately updated in the presence of change, either agent caused or dynamical definition caused. It is this interface which is used by model developers to experiment with the artefact.

The second is a 3D visualisation module developed for the Warwick Game Design society at the University of Warwick. This is a student group who make computer games and at one time it was decided that Cadence, with a suitable set of extensions, could be used to develop games. These extensions involved support for OpenGL windows, animated models, sound effects, sprites, lighting, height-maps and collision detection, as well as for input devices that include the Wii-remote. Much more can be seen of this game library in the next chapter.

4.4.6 Other Extensions

There are other extension modules that have been developed for Cadence. These illustrate the diversity of possible extensions to Cadence and enable Cadence to connect with existing technologies or be used by other software projects in ways that EDEN could not. Below is a list of extensions developed during this work:

EDEN A version of EDEN has been adapted to run as a module of Cadence. There is a communication mechanism between them. More is said on this in §7.2.

XNet Provides a handler for connecting multiple instances of Cadence on multiple machines together in a transparent way by routing events between the machines.

Agents Programmable agency by describing individual actions when certain conditions become true. There is some similarity between these micro-agents and the *actions* found in the ADM [Slade, 1990, p.28].

Web A module that allows web servers to connect to Cadence to provide a web interface. This is currently work-in-progress.