Chapter 4

Cadence: A Prototype Tool

We have seen in chapter 2 how methodologies, languages and modelling tools are all striving for increased flexibility and the ability to evolve software rather than engineer it. All except a few toy environments require that programs, in some form, be developed at the end. To take evolution and flexibility to its greatest possible extent it is clearly necessary to move away from formal programs for some applications. What is hindering this is the lack of an environment capable of supporting production quality software of this informal kind. The main body of this thesis is to explore one possible environment, methodology and conceptual framework that may go some way to allowing end-product capable software without needing a conversion to a program. We call this environment DOSTE and discuss technical aspects in this chapter with examples and development techniques given in chapter 4 and the conceptual framework in chapter 5.

4.1 The DOSTE Idea

- prototype-based objects - dependency - temporal /dataflow style dependency - agents

The Definitive Object State Transition Engine (DOSTE) is the first prototype developed specifically to investigate the question posed by this thesis. The best way to think about DOSTE is as a form of operating system or general computing environment upon which artifacts are developed and agents interact to produce an experience for the user of the machine. The original objective was to produce an operating system that could be explored, experimented with and customised in ways that current systems do not come close to allowing. On a personal note: I wanted access to my machine without needing to write a program to do it. I wanted to play.

There are 3 key concepts behind the DOSTE environment, all of which do exist in some form in a variety of tools. The unique aspect of DOSTE is the way they have been combined and are to be exploited. What is novel about this tool is the simplicity and purity of the object model used, as well as the way in which this has been combined with dynamic declarative definitions to create a hugely rich environment. Numerious different programming paradigms are brought together in an elegant way. Even the concept of computation itself is viewed differently, as graph navigation instead of sets of instructions or mathematical functions. Bringing such an elegant system right down to the machine level is important here. Users can directly manipulate the graph and definitions which are closely mapped to hardware and allows for an large amount of control over the machine, at least in principle. In fact DOSTE should be considered an operating system in its own right and there is a version which will boot entirely on its own. It manages processes, in the form of definition evalution scheduling and agents, as well as dealing with all memory and storage concerns. It is a non-standard kind of virtual machine.

4.1.1 Concrete Objects

The first concept is that of concrete objects as found in prototype or instance-based languages such as Self [ref] or Javascript [ref]. It is important to not have classes and to not attempt to classify objects into specific and restricted types. Properties can be added, removed and changed in any concrete object at any time. This freedom to change is what removes the need to specify and restrict the kinds of data that we are dealing with. We are no longer restricted by the machine to produce formal structures for our formal programs to work with, so instead we can take a more informal semi-structed view.

Semi-structured concepts have been around for a considerable time [ref] and currently, usually, take the form of XML.

- could say stuff about Microsofts failed attempts at formalising this into relational databases. - can also connect with XML and the need for semi-structured data.

4.1.2 Dependency

4.1.3 Agency

4.2 Architecture

Whilst the actual implementation of DOSTE is complex and large, the architecture itself is simple to understand. Over the course of developing DOSTE there have been many variations on how it works, the most significant of which will be described here to justify some of the design decissions. One of the biggest challenges was creating a system that could be adapted easily since a great deal of experimentation was needed to get the scheduling, definition evaluation and agent interaction correct. As a consequence the architecture described here is flexible with almost everything being modular and extendable. The rest of this section will give more detail about the architecture.

The architecture can be summarised as object databases connected together with all communication with the objects being done through events. Definitions are about generating a particular set of events to lookup the resulting object and agents are about sending various read and write events into the system. All events go through the central event processors which effectively acts as a router. The event router can send events to one of many handlers that may even be on another machine. We will now describe the events and event queues in more detail, as well as agents, event handlers and the modular nature of the whole system. The GUI layout example introduced previously will be used here to demonstrate the flow of events and evaluation of definitions.

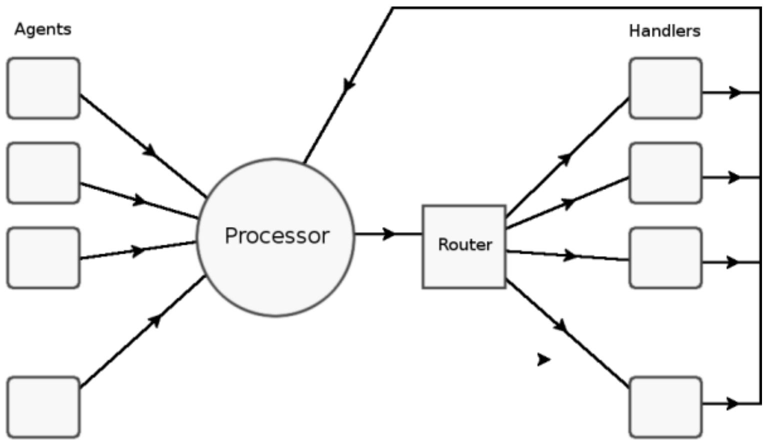

Figure 4.1: DOSTE architecture diagram showing the core components.

4.2.1 Core: 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). Microsoft Windows uses messages and X11 in Linux utilises events for the same purpose. In the major operating systems, however, events are effectively processed by the application polling the OS for them, although in practice the process goes idle if there are no events. In this sense these applications aren't really event-driven and there is still the concept of process. Another possible approach is to have event handlers that get called directly by the operating system. In this model there is no concept of process as such. Linux does, to a limited extent, have this sort of mechanism available in the form of signals.

For our purposes in DOSTE the second model is more appropriate because we have no processes and giving the operating system complete control over event scheduling is vitally important. It also increases the level of concurrency because individual event handlers are more fine-grained than processes. Individual observables and definitions could be thought of as tiny independent functional programs and as a result the amount of concurrency available is far beyond a traditional system. It is impractical to think of each definition as a process which can poll for events so the system must instead be more event-driven. Inter-definition communication must also utilise the event system.

As a consequence of this the core of DOSTE is a massive event scheduler and router with very little else. Events are represented in DOSTE as little 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 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 self-referent definitions. If you fail to correctly order the events you 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 occured and incorrect values are the result.

Events have the following structure:

type The type of event determines how it is processed

destination The object to which this event is being sent

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

result An optional result OID for some events

All events are some action to be performed to a particular object and therefore events have a destination object. It is this destination that determines where the event is sent for processing by a handler. Different objects get managed by different handlers and this is determined by the Object IDentifier (OID). The type attribute determines what action to perform on the destination object. Here are some of most significant event types:

GET Return the value of an attribute within an object. This event is performed synchronously and does not go into a queue for delayed processing. As a result, like all 'get' events, it should only be sent at the correct time, usually inside an agent event handler or when evaluating definitions.

GETKEYS Return a list of all attributes in the object. This creates a new 'buffer' object to store that list and returns the OID of the buffer. A buffer is some temporary block of memory that can be accessed directly without going via the event system itself.

SET Change the value of an attribute within an object to a value given as a parameter to the event. All 'set' events will get added to a queue to be performed asynchronously since it would be dangerous to make that change synchronously in a parallel system. It is also how definitions can evaluate on current values and the changes they generate are not applied immediately.

DEFINE Change the definition of an attribute. Again the definition is given as a buffer object that contains a form of definition byte code which is really just a list of OIDs.

NOTIFY Notify an attribute that its definition is out-of-date and should be re-evaluated at the appropriate time. Again, this event is added to its own queue.

ADDDEP Add a dependency on a given attribute. This will tell an attribute that if it changes it should send a 'notify' event to another objects attribute which is specified in the events parameters.

Agents generate events to observe and manipulate objects whilst the internal dependency maintenance system uses them to add the dependencies and get notified of changes. These events are then sent to be processed either synchronously or asynchronously to the processor module. When they are sent they are added to one of three queues depending on the type of event: write, notify or dependency. Each CPU will go through one queue at a time and process each event. Read events are always synchronous 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 SET 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 worked out. Finally, a notify event will generate a single SET 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 random. Add-dependency events get added to the next queue after notify events so that they are all performed before any SET events. If this was not the case then some SETs 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 complete cycle through the queues is complete the whole process starts again with the first write queue. Each cycle is called an instant.

So now we will look at the button centering example by showing what events take place when certain actions are performed. Figure 3.2 shows the flow of events that is generated as a result of such a definition being made on the x-coordinate of a button. The diagram also tells you which queue the events are added to and shows which queue is currently being processed. It is easy to see that even a single definition event can generate many other events and more complex definitions could generate dozens of

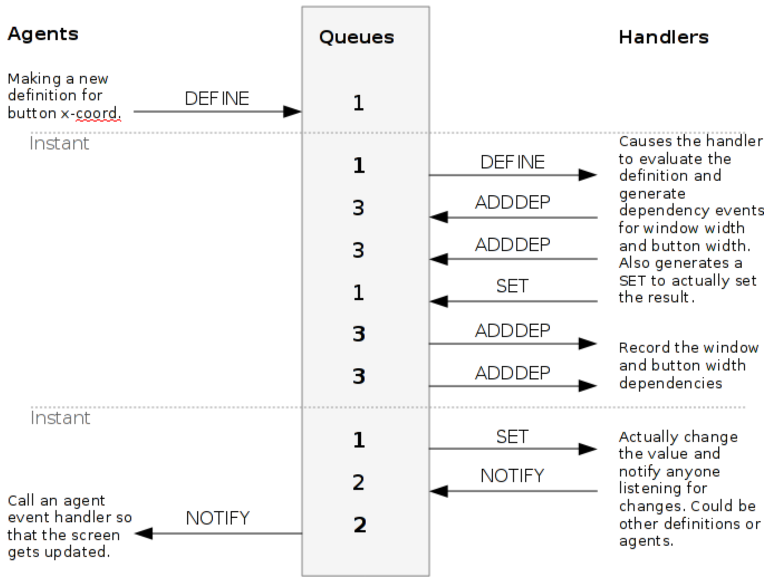
Agents
Queues
Handlers
Making a new definition for button x-coord.
DEFINE
1
Instant
1
3
3
1
3
3
DEFINE
ADDDEP
ADDDEP
SET
ADDDEP
ADDDEP
Causes the handler to evaluate the definition and generate dependency events for window width and button width. Also generates a SET to actually set the result.
Record the window and button width dependencies
Instant
1
2
2
SET
NOTIFY
Actually change the value and notify anyone listening for changes. Could be other definitions or agents.
Call an agent event handler so that the screen gets updated.
NOTIFY

Figure 4.2: The flow of events after an agent makes a new definition.

ADDDEP events. The definition used here subtracts the width of the button divided by 2 from the width of the window divided by 2. It therefore has 2 main dependencies, button width and window width. The example has been simplified because there are likely to be other dependencies such as those on the arithmetic operators.

So the core of DOSTE is a sophisticated and concurrent event processing machine. The next sections identify the modular parts that make use of this.

4.2.2 Handlers

A handler is an event handler that directly receives events for processing. Each handler must be registered with the router by requesting to receive events for a particular range of Object IDentifiers. It has been designed to be modular so that different parts of the object graph can be managed by different handlers because some parts of the graph will be virtual in nature. This enables the integers, for example, to be a part of the graph and hence allows everything to be an object instead of requiring more primitives.

The integers and real numbers are represented by a specific set of OIDs which have all of their events processed by the Number handler. An integer is an object with a specific set of attributes that includes the operators that can be performed on that integer. The operators are also objects which get handled by the Number handler. In this way there is no need to physically store the numerical relationships because this special handler will calculate these operators on the fly and return the relevant objects. To the user it just appears as another part of the object graph. This will become clearer in later sections when we look at the notation.

Another handler is responsible for storing user created objects in memory. It does this using a hash table which uses the objects OID and the attributes OID to find an observable. It should be noted here that object attributes are labeled with OIDs so that the value of an attribute may also be used to select another attribute. This feature is explored further in later sections. Events are routed inside this handler to specific observables which then make the required changes.

Handlers are also responsible for evaluating and storing definitions which are similar to spreadsheet formula to give the value of an observable. They work by describing potentially nested paths through the object graph to get to a result. In order to evaluate a definition you need to generate a large number of read events to navigate through the graph. Definitions are actually stored as objects and so are visible within the object graph.

Handlers can be added and removed at any time whilst the program is running and it is fairly simple to construct new ones with a C++ module. By having such a flexible event processing mechanism as this there is huge potential for extensions to be added in the future.

Here are a few examples of handlers found in our prototype:

Local A local set of objects stored in memory

Network Passes events to remote machines

Numbers Simulates the infinite number objects

IO Represents hardware IO ports

Files Maps the OS file system to objects

4.2.3 Agents

An agent is some entity which observes and interacts with the object graph. It may do this by navigating the graph, changing values or definitions and by creating entirely new objects. Often agents will communicate with the outside world such as displaying something on the screen or taking user input from the mouse. In DOSTE an agent is just a C++ object that can receive notifications from the object graph about changes that occur and it can then navigate and modify the graph.

A large part of the power and flexibility of DOSTE is in the ability to have custom agents added at any time. The developer can write a small agent for a particular task in C++ and then at run-time load it as a module. DOSTE provides a clean and effective API for accessing the object graph and specifying custom event handlers as will be seen in a later section.

Agents are managed by an Agent Handler which is an ordinary handler as described previously. Each C++ agent has a corresponding object in the object graph whose events are handled by the Agent Handler. This allows the ordinary event mechanism to be used for notifying agents of changes because the Agent Handler will, instead of evaluting a definition, call the relevant C++ method when it receives NOTIFY events. There is no reason why different agent handlers could be added to connect with other languages such as Java or perhaps work as a virtual machine for some special agent notation.

Examples of agents can be found in a later section about a game library that uses DOSTE.

4.2.4 Summary

4.3 Notation: DASM

A basic textual notation has been developed as a part of Cadence to enable a user to interact with the underlying DOSTE graph. 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 languages 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 language with which to interact with Cadence other than directly interfacing via C++. 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.

In this section the syntax and semantics of the DASM notation will be given. How DASM relates to the DOSTE graph structure will be illustrated by showing the corresponding graph structures generated when a script is interpreted.

4.3.1 Simple Statements

The most simple form of statement 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 labeled 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. 'this' may also be written as a dot. The following example 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.

```
this x y z
```

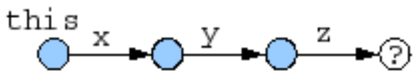

Figure 4.3:

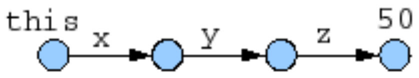

Figure 4.4:

Following from this is the simplest form of assignment where the node an edge points to can be changed. Again a statement must start with a node and one or more edges, however, instead of returning an unknown node a new node is given and the node the edge originates from is returned. Having the origin node returned allows several assignments to be chained together as will be shown later.

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

So now the first example would 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 interpretated as representing a number requires other structures to be in place for numerical operations. These structures and relationships to other numbers have not yet been defined. Therefore it is simply a label with no other meaning as far as the system is concerned.

We may wish to make the edge point to a node from another part of the graph so to do that we put a path on the right-hand-side (rhs) of the equals. It is critically important however that this be surrounded by brackets because otherwise the first identifier will be interpreted as the node it 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.

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

The following is probably incorrect but can be interpreted as suggested in the subsequent example.

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

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

With what has been shown here it is possible to build simple structures and make queries by giving a path through the graph. We have been using the graph terminology throughout but another interpretation is as an object hierarchy where nodes are objects and edges are attributes within those objects. Using this object interpretation makes it clearer how it relates to existing languages but the term object is burdened with additional meaning that is not appropriate here.

Another useful feature worth introducing here are notation 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. All notation variables begin with the '@' symbol as shown in the next example.

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

The above is equivalent to the first assignment example but now every occurence of 'this x y' is replaced by '@xy'. Another benefit of using notation variables is that if the context changes, so that 'this' is a different node, the node stored in the notation variable does not change. In these cases a normal query would not work as expected and these variables prove invaluable.

4.3.2 Numeric and Boolean Operators

In order for a numeral to really be considered a number it needs to be associated with certain numerical operators such as addition and multiplication. These binary operators describe a relationship between numbers that can be described as a graph. It is therefore possible to represent all numerical operators as structures within the DOSTE graph environment and no special cases are needed. In practice these parts of the graph are massive or infinite and so are virtual rather than explicitely defined. However, for purpose of demonstration we will show how these could be given explicitely using the DASM notation.

Lets take addition as our first operator and implement it for the numbers 0, 1 and 2 only. It should be obvious from this to see how it can be expanded for all numbers.

```
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
```

We can clearly see that what we are doing is giving the computer all the relationships between numbers. It is like stating what is true. Unfortunately it is not possible (at present) to give such an example and for the environment to then figure out the remaining relationships. A new piece of syntax has been introduced, the keyword 'new'. All this does is returns a new unique node that has yet to be used for any particular purpose (all edges from this new node point to the 'null' node). Since we have not given

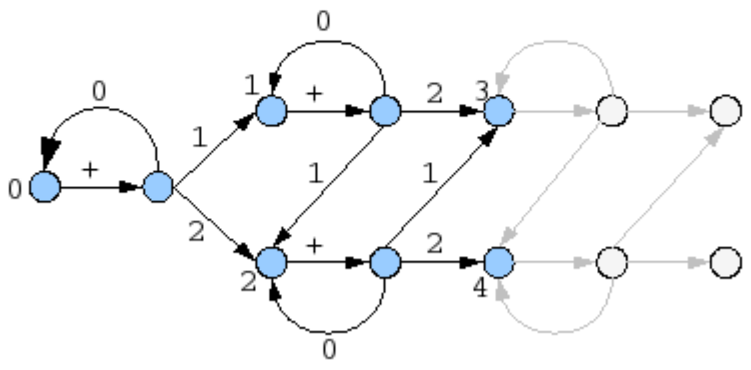

Figure 4.5: A portion of the graph for integer addition.

every relationship the operation given above is not closed or even particularly useful. All examples from this point assume that all operations have been fully defined as above.

Given these definitions it is now possible to perform arithmetic using a query on the graph.

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

Any arithmetic can now be performed as simple graph navigation. There is no computation in the traditional sense but instead the very simple act of following a given path. The agent that does the following is doing the computation and that agent may well be the human user as much as any other machine. In a sense we are going back to the good old days with lookup tables instead of calculators, albeit a big, fast and automated version.

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

Here we see for the first time that it is possible to nest queries so that the result of one of these subqueries can be used to select a path in the main query. For this to work it must be made clear that nodes can be used as edge labels and vice-versa. So the result of a query is a node that is then used to identify an edge from another node. This is a rather unusual but critical feature that makes computation by navigation possible.

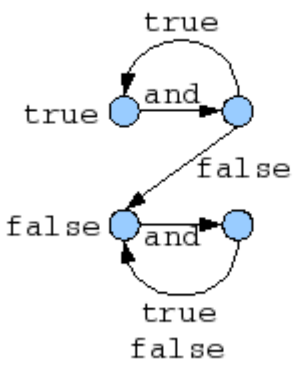

Figure 4.6: Complete graph showing boolean 'and' operator in DOSTE

Much more will be said about the theory behind this in a later chapter. The next, and possibly simpler, task is to define the boolean operators. This can be achieved in exactly the same way and because it is small and finite it can easily be given explicitly so is not a built-in virtual part of the graph. Below we give the 'and' operator.

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

For a quick demonstration we give a simple boolean statement below using only the and operator. Whilst you will be able to work it out in your head it might be fruitful to use the graph to navigate to an answer if it isn't already obvious how computation by graph navigation works.

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

4.3.3 Construction of Objects

Objects are graph nodes and we have seen how to make a new unique graph node. There is not much more to it than that, however, there is additional syntactic sugar and various techniques to help with object construction. This includes cloning, mixing and notation contexts. We also show how large or infinite objects may be specified with a few examples.

The assignment operation in DASM has a specific behaviour with regards to context. It operates a bit like a stack where the immediate lhs, equals and rhs are popped and the action performed. The rest of the left and right remain in place. As a result of this it is possible to chain several assignment operations together without again specifying the subject object. To make this clearer we have given two examples below, the first is fully specified as seen previously and the second makes use of chaining.

```
this x y a = 50;
this x y b = 60;
this x y c = 70
```

```
this x y
    a = 50
    b = 60
    c = 70
```

Both of the above do exactly the same thing. Any amount of white space can be put between each identifier so for readability we used new lines and tabs in the second example. Also note the use of semicolons which reset the context. This is needed in the first example because the context is fully specified on each line whereas in the second example the context does not change from one assignment to the next.

Although the above can be useful it is even more useful when combined with the 'new' keyword as a means of constructing a new object. The following example makes a new node (object) and proceeds to specify several edges from that node. The newly constructed object is then pointed to by the 'button' edge.

```
this button = (new
    x = 10
    y = 50
    width = 100
    caption = "Press␣Me"
)
```

The above demonstrates ex nihilo construction of an object but another approach is to clone some existing object and then modify it accordingly. Instead of simply copying the code as you would do in EDEN it is possible to use the keyword 'union' to internally perform the copy operation. When 'union' is combined with 'new' it has the effect of making a copy. In the example below we make use of a notation variable called '@prototypes' which contains a collection of pre made objects.

```
this button = (new
    union (@prototypes button)
    x = 100
    y = 50
    caption = "Copy"
)
```

The 'union' operator works as an infix operator and can be chained together to merge several objects into a single object. The result of the 'union' operation is the same object as on the lhs.

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

4.3.4 Simple Definitions

So far we have looked at building static structures to represent state but in addition to that we can give declarative definitions to describe how these structures should change in the presence of other changes caused by agent action. In this way we can give dependency relationships to DOSTE which it can then automatically update. These definitions are similar to spreadsheet formula.

The simplest definition acts as a shortcut so that a graph query is used to determine where an edge points and if some agent changed the graph such that this query gives a different result then this defined edge will also change to that new result.

```
.a = 5;
.b is { .a }
```

so when queried 'b' will give the node '5'. If 'a' was changed to '6' then 'b' will also be '6'. The keyword 'is' is used to mean definition and the definition itself must be enclosed in braces. The definition can be any graph query so it can be any calculation. The next example shows a simple calculation query used as a definition.

```
.c is { .a * (.b) + (.a) }
```

4.3.5 Conditional Statements

Any computer language will allow for conditional statements, DASM is no different. However, in DASM there is no need for an explicit conditional construct because such statements can be achieved using the graph.

```
.if = (new
    cond is { @root a == 0 }
    true is { @root b }
    false is { @root c }
    result is { .(.cond) }
)
```

The above example is an if statement built entirely from an object with definitions. The 'cond' observable is some boolean used to select a result. The 'result' observable contains the relevant selected value. This can be a little tedious to write so there is syntactic sugar available:

```
.result is {
    if (@root a == 0) {
        @root b
    } else {
        @root c
    }
}
```

Having this sugar makes it easier to combine multiple if-statements either in a nested fashion or using 'else if'. Not only do we have 'if' but also 'select' which is similar to 'switch' in C. There are no type restrictions as such with a 'select' statement, so it is only different to an 'if' in that it can be more than true or false.

```
.animal = cat;
.label is {
    select (.animal) {
    cat: "Cat"
    dog: "Dog"
    rabbit: "Rabbit"
    : "Unknown"
    }
}
```

4.3.6 Iterative Definitions

The exact semantics of definitions in DOSTE can be shown by the following:

```
.a = 0
.a is {0}
.a := {0}
```

The first example will make 'a' be 0 right now and, unless some other change is made, it will stay as 0. The second form says that 'a' will always be 0 regardless of any assignments made to it. Only changing the definition can change the value. The third example says that 'a' will become 0 in the very next instant but right now it could be anything. So doing .a = 1 would temporarily make 'a' be 1 but in the very next instant it would go back to 0 because of the definition. Each observable has both a current value and a definition to say what that value will become.

Once you have given an observable a definition you can effectively remove that definition by giving it a new definition which refers to itself. This means that it will always be whatever it already is. Note, however, that you cannot do the same using 'is' since that will produce a cyclic definition that cannot be evaluated.

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

Equally though it is possible to say that 'a' is what it already is plus something. The following example implements a counter which counts as fast as the DOSTE environment can and starts at 0.

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

4.4 C++ API

We will briefly discuss the C++ API constructed for DOSTE since this is a key part of it's flexibility and usefulness. 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 DOSTE it is also possible to add these new agents at any time while the environment is running.

C++ comes with a large number of features for customising the language. This includes operator overriding and macros that are used extensively in the API. An agent in DOSTE is an object in C++ which can respond to changes that occur in the object 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 object graph using a simple mechanism as well as use any other C++ libraries to, for example, draw graphics on the screen.

4.4.1 Modules

A module is a dynamically loaded C++ library that can contain DOSTE agents or handlers. Each module has three functions. One to initialise and register the agents that the module contains. One which is called on a regular basis at a specified frequency. This update method is used for those tasks which need to be performed independently of DOSTE event handlers such a OpenGL 3D rendering. Finally there is a method to cleanup when the module is unloaded.

Modules are actually loaded and managed by a built-in DOSTE agent that watches for changes in an object and will load a module if a new module object is added to it. Custom module loaders could be added by writing new agents to perform that task. One such module loader might use Java class files as it's source of new agents.

4.4.2 Agents

An agent in the C++ library is an object in C++ that is an instance of a class that inherits the Agent virtual class. An Agent object is mapped directly to a DOSTE object so that this agent can directly access, modify and respond to events for that object. A good example for this might be a button object in DOSTE which will have an associated C++ Agent object to listen for changes to that button and update the screen accordingly. Each button will have a different C++ object behind it but all from the same class.

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 DOSTE can look at a DOSTE object and determine what type it should be. When another agent requests an agent object from a particular DOSTE object 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 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. This enables classes to freely communicate without knowing anything about each other. Hence there is increased flexibility.

There are many libraries available for C++ which provide an event-based mechanism. Such libraries include GUI libraries such as Qt. Often the approach taken here is to preprocess the C++ source files to convert special event syntax into real C++. Alternatively complex use of macros and templates can be abused into a similar role. DOSTE makes extensive use of macros and templates to provide a user friendly means of creating agent event handlers. Each handler is a C++ procedure in an agent object which is associated with some definition that selects which observable in DOSTE this handler is observing.

4.4.3 Handlers

Similar to an agent, a handler also needs to be registered but for a particular range of Object Identifiers. A handler is another class that can be specialised for a particular purpose. It has a single virtual method which receives events from the processors. Whilst the API is simple enough the custom classes must provide certain functionality. They must process all events and return correct results. This may involve correctly processing definitions and sending additional events. Exactly what is required will obviously depend upon the nature of the handler.

4.4.4 OIDs and Definitions