Third Year Project Report - "Re-implementing the Abstract Definitive Machine"

David P. Cunningham
Supervisor: Meurig Beynon

2003-2004

Abstract

The Abstract Definitive Machine is analysed and criticised in a contemporary Empirical Modelling context. Certain ADM ideas are developed into a new Empirical Modelling environment, a definitive notation with a radical nested structure and a finer grained style of interaction, designed to solve some of Eden's problems. New questions are raised about the presence of static type checking in Empirical Modelling tools, and the role of the tool's interface in presenting a graphical depiction of the model. Suggestions are made as to how more of definitive scripts' limitations can be overcome.

Keywords:

- ADM
- Asylum
- Hive
- Shiver
- Dependency
- Empirical
- Modelling
- Library
- Interpreter

Contents

Chapter 1

Introduction

The original Abstract Definitive Machine implementation "am" was developed by Slade, while he produced his Masters' thesis[4]. This implementation has since been vastly superseded by Eden, in terms of functionality *and* maturity. It *is*, however, based on ideas that are closer to Empirical Modelling concepts than Eden has ever ascribed. Eden is a great improvement over both imperative and declarative languages, in terms of implementing Empirical Modelling concepts, but as a definitive notation, it does not directly support certain classes of models. For example models involving the interactions of different agents, are hard to represent (although this can be achieved with a disciplined naming strategy, or use of dtkeden). Especially, it has poor support for modelling situations where there is a prescribed passage of "time".

It was felt that a re-implementation of the ADM might at worst provide an account of the usefulness of ADM ideas, many years on from their initial conception, and at best lead the way for a next generation of Empirical Modelling tools. As well as doing justice to the ideas developed by Slade, we can use this opportunity to study some of Eden's problems, and how they can be overcome. In many ways the re-implementation has been a proof-of-concept implementation of ideas that were never present in the original ADM. Some of these ideas are a natural evolution of ADM ideas (such as nesting instances), some are a slightly different interpretation of the same

vision (temporal dependency) and some ideas are brand new (static type checking and graphic visualisation of models).

In all of these cases, development of the Asylum (as the new implementation has now been named), has provided an opportunity to investigate these ideas and test their usefulness in a concrete tool. Because time was a major constraining factor, the amount of implementation work that could be done was quite limited. However, the thinking and design work behind the implementation is more complete, and it is still useful to review the conclusions made during this process.

The structure of this report is as follows: First we summarise the features of the ADM. This provides a base on which we can discuss, criticise and extend these features in a contemporary setting, with reference to the more recent developments in Empirical Modelling and Definitive Notations. Following this theoretical discussion is an account of the actual implementation of the Asylum in C code, and all the pitfalls and problems that the implementation process presented. Finally we discuss how the Asylum is still limited, and what possible steps could be taken in the future, in order to increase its effectiveness as an Empirical Modelling tool.

Chapter 2

The ADM according to Slade

It is beneficial at this stage to briefly summarise the features of the ADM that make it different to Eden. A more complete description, formed from first principles, can be found in Slade's thesis[4]. The following description is a direct interpretation of the latter, but focuses on specific details that are felt to be most important.

First we must understand that the purpose of the ADM is to provide a vehicle for the animation of LSD scripts. LSD is a specification language whereby we can account for our understanding of the way a set of agents interact in a model. By "animate", we mean we want to see and interact with the model in order to understand it better. The syntax of the ADM is therefore very similar to LSD, having been designed to allow the easy conversion of LSD into a form the computer can interpret and animate.

2.1 Agents

The first interesting feature of the ADM is therefore the way that the model is divided into a set of entities, or agents. The term "agent" is the more abstract definition used by LSD. The term "entity" is used to describe the feature of the ADM that is used to implement agents. An entity in the ADM describes the behaviour of an object within the model. Any object can be modelled in this fashion, no matter how complex, or significant a role it

plays in the model. An entity can represent the concept of a phone, a car, an ignition system, or a driver. These objects are concurrently interacting within the model.

Typically an entity owns some observables and has control over them. These observables will change value over time, this being the representation of the entity's *activity*. An entity will also require information from some other instances of entities in order to function properly. This is the representation of communication between the various agents in the system.

Each entity can be instantiated once, or many times if there is a way of keeping the instances distinct (more on this in section 2.4). Each instance has its own state and makes its own independent decisions, but the manner in which it behaves is determined by its entity. Each instance of an entity can reference observables in other instances, to receive information.

There are two immediate consequences of this feature. Firstly we can build models with different agents that interact (rather than a flat set of observables), much in the way that we can perceive a real world situation to be a collection of interacting agents. This is not particularly useful, since it is just a syntactical structure that makes our models more clear. The second is that models can be constructed with a set of identically behaving agents interacting in a prescribed way. The following is a demonstration of this claim. It is a model of two players playing the traditional game of "Paper Scissors Stone", but the "Player" entity is defined only once. This is important because in Eden, there is no way to instantiate multiple collections of observables with an associated prescribed behaviour.

```
entity player (_id,_opponent) {
DEFINITION
        choice[_id],
        diff[_id] = choice[_id] - choice[_opponent],              # <--- note
        win1[_id] = diff[_id] + (diff[_id]<0) * 3,
        win[_id] = win1[_id] == 1,
        lose[_id] = win1[_id] == 2,
        draw[_id] = !win[_id] && !lose[_id]
ACTION
        true -> choice[_id] = |rand(3)|
}
player(0,1)
```

```
player(1,0)
```

Note how the definition of an arbitrary player's "win" status refers to the choices randomly made by both players, hence we are representing two communicating agents with the same behavioural specification.

2.2 Observables: Oracles and Dependency

k An entity's observables can be either an oracle (information brought in from some other instance, and hence read-only), or a derivate (information defined locally, based on other observables). The derivate is, in general, defined to be the result of a function applied to a set of other observables. The usual infix operators are allowed such as `+`, `-`, `*` and `/`, as well as logical operations such as `&&`, and `||`. Strangely, the declarative if statement `?:` seems to be missing. Perhaps this is an oversight in the original implementation. Dependencies allow some basic computation and shaping to be done to the oracles, before decisions are made based on them.

It is significant that the notion of dependency is a very direct metaphor of the manner in which we perceive the relationships between different objects' observables. It is therefore an ideal mechanism through which the modeller can express their experience of interacting agents, while retaining the ability to "compute" the construal in reasonable time. Much can be said about the power of dependency in Empirical Modelling, but this is not of much interest here, as we wish to investigate the features of the ADM that separate it from Eden, which has the same mechanism of dependency as was just described.

So far our entities are defined like so:

- The entity name.
- A set of observables that are imported from other specified instances.
- A set of observables that are functionally derived from other observables.

Here is another entity that makes clearer use of these features, than the "Paper Scissors Stone" player defined in section 2.1:

```
entity student ()
{
DEFINITION
    overallpercent = catsreceived / ((120+catstaken)/2),
    catstaken = 135,
    catsreceived = projectmark*30 + exammarks * (catstaken-30)
}
```

The oracles in this example are `projectmark` and `exammarks`. The idea is that these names exist in some other instance of some other entity. If these observables are unavailable, they evaluate to "@", an exceptional value that denotes the potentially undefined nature of observables. This value spreads through functional expressions effectively preventing any evaluation of that observable, and thus we can deal with models that are not completely built. This mechanism can be thought of adding an element of safety to the evaluation of dependencies, and also as allowing a construal with incomplete knowledge, to still be represented in the machine.

Expressions also evaluate to "@" if there is a cyclic dependency in these derived observables. This, again, acts as a placeholder for the value of an observable that cannot be calculated.

Clearly a "well defined" instance can thus be visualised as a simple directed acyclic graph, showing the flow of information through the computations. In itself, this does not allow us to express much character within these entities, as they have no way of enumerating a state. We refer here to "state" in the automata sense of the word, where state is the result of repeatedly applying a state transition function to an initial state. This is analogous to the limitations of combinatorial logic circuits.

2.3 Execution Cycle

To allow us to represent more complex entities, the ADM allows another mechanism within its entities. It is possible to specify an action that occurs

under special conditions (i.e. when a boolean expression on a set of observables holds true). The action can be a redefinition of a derivate observable, an instantiation of an instance, or a deletion of an instance.

A redefinition allows the function expression that defines the value of an observable, to be indefinitely changed to another function expression, potentially of a different set of observables. This also allows the changing of an expression such as a literal number, e.g. "3" into another literal number. It is additionally possible to evaluate an expression into a literal value, and embed this value into the new definition. This is possible with the `|exp|` syntax.

An instantiation of an entity is equivalent to creating the set of derived observables owned by that instance, with their default values. A deletion of an instance is equivalent to undefining every derived observable owned by that instance.

The ADM processes these actions with the following "execution cycle":

1. The ADM is in a steady state.
2. All guards are evaluated and a pending set of actions is built up.
3. These actions are concurrently performed.
4. The ADM is in a steady state once again.

Because the actions are concurrently performed, there is some possibility of race conditions arising, if two actions concurrently redefine the same observable (including during the activity of deleting or instantiating the containing instance), or if an observable is evaluated at the same time as it is modified. The ADM refers to the resultant erroneous state as a "singular state" and assigns the "@" value to all affected observables.

A simple entity which demonstrates actions is a ticking clock:

```
entity clock() {
DEFINITION
    seconds = 15,
    minutes = 15,
    hours = 3
```

```
ACTION
    seconds<59  -> seconds = |seconds+1|,
    seconds==59 -> seconds = 0,
    seconds==59 && minutes<59 -> minutes = |minutes+1|,
    seconds==59 && minutes==59 -> minutes = 0,
    seconds==59 && minutes==59 && hours<11 -> hours = |hours+1|,
    seconds==59 && minutes==59 && hours==11 -> hours = 0
}
```

When the guards are evaluated, anywhere between 1 and 3 actions might be performed. Here is a transcript of this script being performed in the original ADM implementation "am". We can see that after 10000 execution cycles, the clock has increased by approximately 3 hours.

```
compiling clock()
./am> clock()
instantiating clock
./am> set iterations = 10000
./am> start
...
* 10000 iterations successfully completed
./am> l ds

               DEFINITION STORE
               ****************
Variable # 1: seconds = 55
Variable # 2: minutes = 1
Variable # 3: hours = 6
```

This mechanism is important, because it allows us to represent the changing state of the hands of the clock, over time. This kind of "temporal dependency" is not something that is possible in Eden, without using a while loop. With some imagination, it can be seen that we are specifying the rate of change of the clock hands, like in a differential equation. This mechanism is closer to the reality of what is occurring, and is therefore more ideally suited as a fundamental element of an Empirical Modelling tool.

2.4 Parameters

As stated earlier, it is possible to instantiate an entity more than once. This is useful because often a model will have multiple "beings" that are very

similar in their behaviour, but are existing in different circumstances. Since an instance has its own state, and triggers its own actions, this is a useful mechanism to have when building a construal.

Because there is only one global namespace for observables, clearly there will be a conflict if an entity is instantiated twice, so how can we get around this? The ADM allows instantiating of an entity with a specific set of parameter values. This is roughly analogous with constructor arguments in object orientated languages. The idea is to give instances (and hence their observables) some information that can be used to identify them amongst other observables belonging to a different instance. This is best explained with an example:

```
entity human (_id,_age,_male,_mother,_father) {
DEFINITION
    age[_id] = _age,
    male[_id] = _male,
    parents_age_difference[_id] = age[_father] - age[_mother],
}
human(0,24,true,1,2)
human(1,50,true,3,4)
human(2,44,false,5,6)
```

Here, we instantiate 3 Human beings. The Human observables can be told apart because they are referenced using an index value. An instance can reference its own observables by using the `_id` parameter as an index. An instance can also access an arbitrary observable by using some arbitrary index. The other parameters give the entity some information it can use to derive its default definitions. In this case, the parameter `_age` is used to initialise part of the instance's state, but the parameter `_father` is used as an identity for contacting an observable of another instance. The essential lesson here is that an "`_id`" parameter can be used to uniquely identify an instance's observables, and therefore it can be used to allow multiple instances of the same entity.

This is an interesting strategy, most other languages have used specific placeholders for referring to different instances. For example in Java and C++ you give each instance a name, in the form of "`Object myObject = new Object();`". However it does work, and it seems appropriate to identify

objects by the parameters that describe them, since this is how we typically identify anonymous objects in the real world. For example "the chair in the corner" or "the red car". The problem is that this can get confusing, and if we are to use a parameter for this purpose, we would rather it were a piece of text, which is far more descriptive and readable. If we want, we can still name the object after its parameters, E.G. "`my_red_car is Car(RED)`".

2.5 Criticism Of The "am" Implementation

Why, despite the closer relationship that the ADM has with Empirical Modelling, has Eden grown to be the dominant tool? There is a converter between ADM scripts and Eden code, (although this does not allow for the interactive style of model building, which is such an important property of EM tools) which has been used successfully in FootballTurner2000[9]. This suggests that the ADM has enough expressive power to construct sophisticated models of interacting agents. It suggests that if there were a good implementation of the ADM, it would be well received by the modellers who currently use Eden. So what is wrong with the original ADM implementation "am"?

When we try to use "am", it quickly becomes apparent that the software has major usability problems. Because interaction is an important feature for Empirical Modelling tools, the fact that "am" is a console application is an immediate flaw. Command line interfaces are flexible and fast, but they cannot easily convey a large amount of structured data. Having said this, Eden is essentially a command-line application, even in tkeden we still enter commands in a special language, into a window to be interpreted by the tool. Interaction with the ADM is somewhat more structured, however. Since we are dealing with an enumerating state, and the model is divided into multiple agents, rather than just a list of functions and observables. We really *require* an interface that presents the current "state" of the model in a clear manner. However even with the low expectations that we may have for a command line interface application, "am" is remarkably hard to interact with.

The "am" program code uses a flex/yacc parser generator combination to

process the modeller's input. This seems like a good idea at first, it factors away the gritty details of receiving input from the console, and allows us to build up expressive grammars. There is one major problem however: When we interact with an Empirical Modelling tool, it is important that there is a continual process of evaluation, comparison, and redefinition made by the modeller: The modeller must observe the behaviour of the model (state as experienced) and compare it to the behaviour of the referent (as experienced or imagined). The modeller must then identify where the current model is inaccurate, and amend it by making redefinitions.

In the case of the ADM, this means making modifications to the entities, and observing the interactions between agents over the passage of several execution cycles. The essential point is that each modification has to be very small, and each observation made of the current state of the model must be quick. The tighter we can make the loop of evaluation and improvement, the more effective the modelling process will be.

In contrast to the tight loop of interaction required, building expressions for the "am" interpreter is a lengthy process, since an entire structured entity "clause" has to be typed out before anything can be observed. A precise sequence of tokens must be formed before the meaning of the interaction can be understood. This is simply the nature of parsed grammars. This deficit is not surprising when we consider that Slade was building a proof of concept implementation for his machine. It was of major concern to test the expressive power of the language, rather than building an interactive tool (which is actually a difficult task to accomplish well).

Adding to the problems faced by the user, who has to provide a lengthy, structured sequence of tokens, the currently running instances have to be deleted and re-instantiated when a change is made to the entities from which they are derived. The mode of interaction within "am" is much more akin to the process of programming, compiling, and test, that we associate with conventional programming. This can be contrasted to Eden, where observations can be made at any time during "execution" of a model, and changes made to both definitive and procedural code.

I believe that this process can be made even more rewarding than Eden al-

lows, by reducing the scope of modeller modifications to the smallest possible unit, and having direct and immediate visual feedback after each interaction. This means having a real-time graphical display of the model, including the state of entities and the interacting instances. This also means turning the redefinition process into a sequence of extremely short actions, the result of each action being immediately presented to the modeller. Rather than creating an entity all at once, we must first create the entity, then spend some time working on its internal definition, maybe with some interacting instances already alive. We should also be able to backtrack at any point, it must be possible to undo any work that has been done.

This style of model building is as far removed from traditional programming as we can imagine. It has a lot more in common with spreadsheet applications, and even what we might call "drawing" programs: Programs that feature a canvas and immediate user feedback during edits. During the Asylum's design process, I have strived to make this possible. This will be a major topic of discussion in section 6.1.1.

Structured language may *not*, after all, be the best way of communicating our imaginative, intuitive, creative processes to a computer. This may have been the only possible method when computers were very slow and graphical displays had not yet been developed, but with modern technology, we can investigate the power of symbolic visualisation and feedback in the creative process of developing computer software. Language has always been a tool for communicating and recording information, and may not be best suited as a medium for interactively building a construal.

A study of the programming code that defines "am" shows that it is written in non-standard C, pre-ANSI. This is a portability problem, and in the future it may be very difficult to compile this code. The implementation is also rather poorly documented. These are issues I have attempted to avoid while developing the Asylum.

This concludes the treatise of the major features of the Abstract Definitive Machine, as designed by Slade. What follows is criticism and development of these ideas, with respect to the problems faced by Eden and what we now know to be the requirements of Empirical Modelling tools.

Chapter 3

Temporal Dependency

As mentioned in Chapter 2.3, a great many models require the passage of time to have a significant role in their behaviour. Models such as the Vehicle Cruise Control demonstration[10], which represents the behaviour of a system that drives a vehicle at a constant velocity over a hill. In Computer Science terminology, what we are simulating here is a transition function that decides the future based on the current state. If s is the current state, then the future state s' can be calculated by application of the function f on the current state s:

$s' = f(s)$.

In physics and engineering, we often use a definition of change in order to model dynamic systems. For example when we mathematically model the temperature of a block of metal as it radiates its heat into its environment, we might use a differential equation like so:

$\frac{dT}{dt} = -kT$

By this we mean that the rate of change of temperature is related to the current temperature. This is the kind of relationship that we have to represent in a construal of a dynamic model. In general, we can experience the relationship between the current state and the next, and represent this with a dependency of the form `newstate is f(currentstate)`. We must make the distinction here between this "animation" state, a measure of temporal progression and the result of the application of many state transitions since

the model began, with other definitions of "state" used in Empirical Modelling. The notion of "state as experienced" describes the current state of the construal, the definitions and instances currently in place, rather than the union of all the observable data.

To elaborate on this last point, we can examine the Vehicle Cruise Control model. The animation state includes the position and speed of the car. The state as experienced is the collection of dependencies that the modeller has defined, that are the reason the car in the model behaves the way it does.

3.1 Study of ADM Representation

The execution cycle introduced by the Abstract Definitive Machine aims to provide a system of prescribed animation state transition. We are aware of a discrete time step in the model, and by prescribing actions that concurrently occur between time steps, we are specifying the transition from one steady animation state to the next. There is however some redundancy and syntactic sugaring in this notation, and it is instructive to strip this away, to discover the most basic definitive notation that can describe a dynamic model. This is useful because by pushing as much of a model as possible into the `DEFINITION` section of an entity, we can see what aspects of execution cycles can already be represented in Eden. When we find we can no longer represent the power of actions with definitions, we will know what features Eden lacks.

First we can study the nature of the possible actions that can be performed. There are three possible actions: redefinition of an observable, instantiation of an entity, and deletion of an instance. Deletion of an instance means to render its observables undefined. Since we can simply redefine all of its observables to "@", this is not necessary. Instantiation of an entity that has previously been deleted, is thus simply a matter of redefining these observables to their "active" definitions. If we want to instantiate a brand new entity, then our model is a construal of a system where agents are completely dynamic. This seems like a good metaphor of reality, where people can enter or leave a situation, however there are criticisms of this technique

when represented on a computer.

The tool must present a visual representation of the current state of the model. This is very hard to do if the model is constantly changing shape, and worse, it is very hard for the modeller to perceive the behaviour of the model, when his only point of reference is the inconsistent visualisation. It would be far better to have a consistent model "shape" and change the values of the observables over time. This argument is analogous to the argument that self-modifying computer programs are much harder to debug than programs with a static structure.

This is not to be confused with the action of deleting and instantiating performed by the *modeller*, which is a crucial part of the modelling process. We are instead referring to the animated behaviour of a model, i.e. its defined automatic behaviour, that the modeller must observe.

If we do not allow dynamic deletions and instantiations within actions, that leaves the execution cycle to be a simple set of guarded redefinitions of observables. Generally, a redefinition of an observable means discarding the old functional definition and replacing it with a new definition. Trivially, this means evaluations after the redefinition result in a different value.

Consider, instead of changing the definition from "`o is f1(S1)`" to "`o is f2(S2)`" during an action, if the definition was originally of the form "`o is x?f1(S1):f2(S2)`", with the initial value of x being `TRUE`. Then the action needs only redefine the literal value of observable x to `FALSE`, if a guard is true.

Now our execution cycle is simply evaluating a set of guards, and for those guards that evaluate to `TRUE`, assigning a particular value to an observable. This value will then *select* the required dependency, using a definition. Since the guards have referential transparency, this can be represented as follows:

```
guard -> obs1 = value
guard -> obs2 = value
guard -> ...
guard2 -> obs3 = value
guard2 -> obs4 = value
guard2 -> ...
...
```

Since if the guard is false, nothing happens, we can represent this as follows:

```
guard -> obs = value
!guard -> obs = |obs|
...
```

This can then be trivially re-arranged to:

```
TRUE -> obs = |guard ? value : obs|
        ...
```

We can factor out the expression into a definition of the form: "`obs' is guard ? obsnew : obs`", and that leaves the execution cycle to simply be a set of assignments of the form `obs := obs'`, where obs' has been defined to be a function of obs. This is directly analogous to the technique described at the beginning of this chapter. It is also slightly better to express the change of state with dependency, rather than imperative actions, because we can observe what the next state will be (by evaluating `obs'` before it is set to `obs`) and also the reason that `obs'` has the value it does, by tracing back the dependencies. There is also the issue that the modeller should make definitions that relate to his universal experience of the referent. If those definitions are being replaced in order to represent animation, then the state as experienced is being mixed up with the animation state of the model, which might lead to confusion.

3.2 Study of Eden Representation

We have discovered that the fundamental power of actions, is in regular prescribed observable reassignments. This maps very well to a technique often used in Eden models to simulate the passage of time. Observe the following transcript of an Eden session:

```
~empublic/bin/ttyeden
1|> state = 0;
2|> newstate is state + 1;
3|> while (1) { state = newstate; writeln(state); }
1
```

```
2
3
4
...
```

We have defined the state transition function, and an initial state of 0. The while loop is providing the functionality of the execution cycle, that of continually evaluating "`newstate`", and setting "`state`" to be that value.

Although this works, it is hard for the modeller to be involved in the process. Ideally we would want to set the rate of execution cycles, and also be able to manually step through an execution, one step at a time. We can see however, that the expressive power of an execution cycle is being used, under the guise of Eden's lower level procedural features.

3.3 Clocked Instances and Evaluation Dependency

The approach taken in the Asylum is analogous to the techniques used in sequential logic circuits.

The first idea is like conventional Eden definitions, and the `DEFINITION` part of ADM entities. Observables are linked with functional definitions to create an acyclic network of dependency. This can be represented with a graph of nodes and directed edges, the nodes representing observables, and the edges representing dependency. This is a form of synchronised data-flow, however we can observe the flow of data, compare with a referent and interact with the graph to change its structure. This is an iterative process of trial and improvement. If an edge is removed, the observable is undefined, in the same way manner as in the original ADM.

Because some dependencies have more than one "operand", we group dependencies of this form with boxes. This is more clearly illustrated in figure 3.1, which can be thought of as a "zoomed in" view of a set of definitions. The observables are to the right of the boxes, where the arrows start. The arrowheads of the dependencies point to the values "go in" to the operator to be computed. Operators in dependency functions can be thought

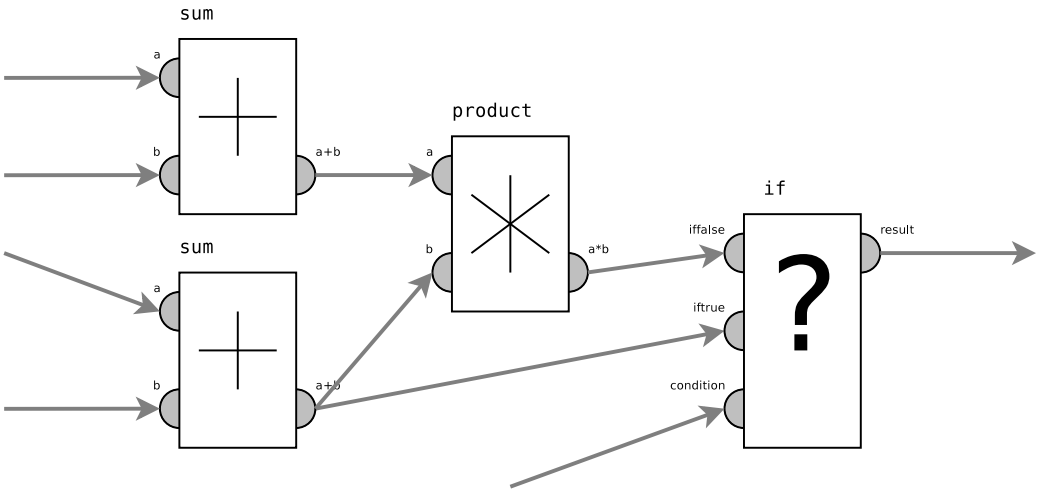


Figure 3.1: A dependency graph with operators represented with instances.

of as instances in their own right, with oracles, and an observable that is "advised".

In order to have the future state of an observable depend upon the current state, we need to add a loop into this graph. We need to make it clear that this loop-back edge is there to define animation state transition, rather than a dependency in the conventional sense. Thus we have two kinds of dependency. Directed dependency is propagated automatically, and thus the modeller is never aware of any state of the model where the dependency is not satisfied. Evaluated dependency means the operation simply reads the value at the time the operation is calculated.

Evaluation dependency allows us to build models such as this integrator: "`integrator is x + |integrator|;`". When `x` changes, it is added onto the current value in the integrator. This assumes that `x` is a source of agency in this model, and the model is reactive in the way that it deals with this agency. This can be thought of as reactive animation state, and could be useful when implementing certain UI features on top of a model, such as push buttons.

There is also a mechanism that re-implements the ADM's execution cycle. If an operator's operands are all evaluation dependencies, E.G. "`a is |b|`

`+ |c|`", then that operator can be "clocked". During an execution cycle (or in the language of digital electronics, a "clock cycle"), all clocked operators evaluate and compute their oracles. This is a source of agency since the change will propagate down the directed dependencies.

There is one feature we yet need, and that is the ability to have an initial state. Nodes that have an animated state (i.e. nodes that have evaluation dependency edges leading into them) must have a well defined initial state. This is allowed in the ADM because the definitions in the `DEFINITION` section of an entity clause are taken to be the initial observable values, before any execution cycles begin. If the initial state is undefined, then all the states after this initial state will be also be undefined.

As with the ADM, there is a possibility of race conditions arising in models that are defined in this way. A race condition occurs where an observable can be evaluated while it has not yet been updated properly. This has to be prevented, in a similar way to the way circular dependency has to be prevented. Typically, it seems that if a directed dependency will cause circular dependency, then evaluation dependency should be used instead, and if an evaluation dependency will cause a race condition, directed dependency should be used instead. Figure 3.2 shows the simplest possible models that demonstrate circular dependency or a race condition.

In the presentation[3], there was reference to a kettle model, that modelled the temperature of a kettle over time. It showed how the use of a thermostat that turned on a coil if the water was below 80^oC, but turned it off if the temperature was above this level, would cause the temperature to stay at approximately 80^oC. A diagram of this model is included here (figure 3.3) to demonstrate the application of evaluation dependency in a more complex model.

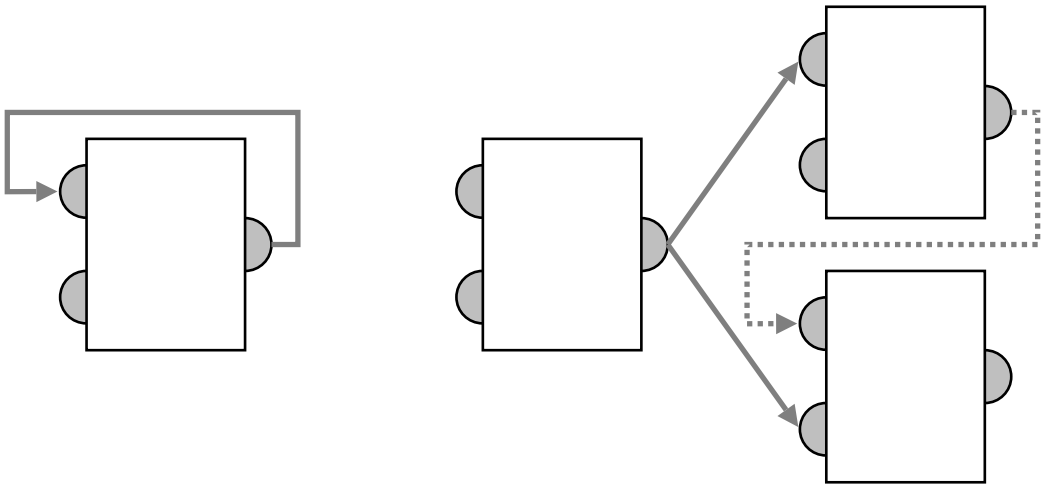


Figure 3.2: Circular dependency (left) and a race condition (right). Evaluation dependency is represented with dotted lines, and directed dependency with solid lines.

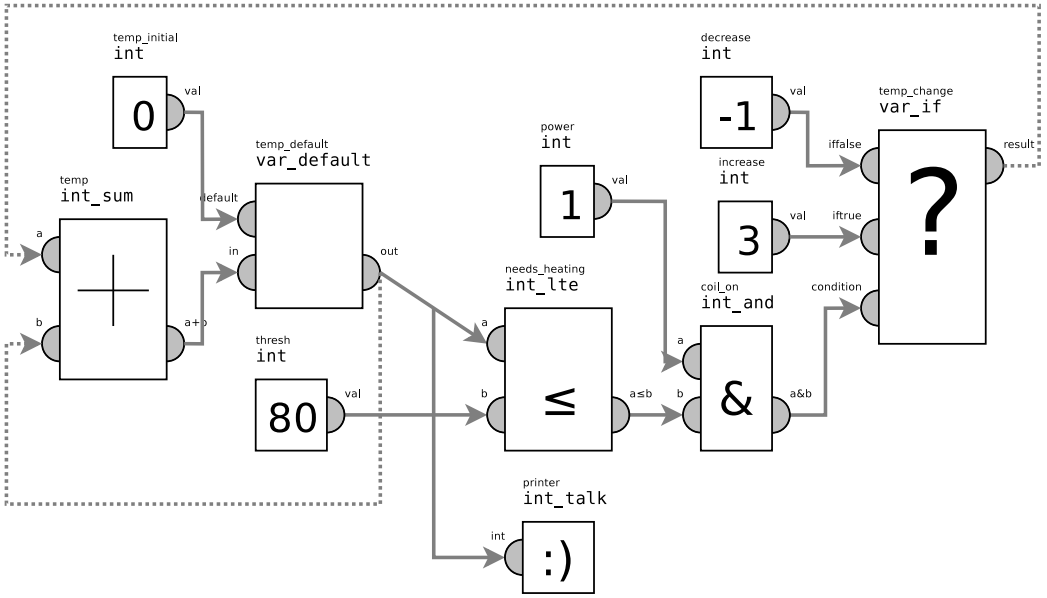


Figure 3.3: Model of a kettle. Note the use of the "`var_default`" instance, to give the loop an initial state.

Chapter 4

Nesting Instances

This is maybe the most successful feature of the Asylum implementation. It aims to solve some of the problems present in Eden, with inspiration from the ADM. The basic idea is that since operators can be thought of as entities, and applications of these operators can be thought of as instances, that an entity defined by the modeller is really defined in terms of a set of nested instances linked together with dependency. So far this is just an interpretation of the ADM's mechanism, but we can extend it by saying that a "nesting entity" can be defined in terms of any instances, not just instances of predefined operations, but other nesting entities. A special root entity is defined, which has one instance, the root instance, and in this root entity can be placed instances of other entities.

4.1 Comparator Example

Figure 4.1 shows a model of a comparison network. The network contains 5 comparators, enough to sort 4 numbers. The comparator entity itself is defined separately in a nesting entity, and instantiated 5 times. In order for this to work, the instances are dealing with different information in their different situations. Just like with the operations, or "predefined entities" such as `int_min` and `int_max`, nesting entities have oracles and advice. The oracles act as an input to the entity, and the advice allow it to export infor-

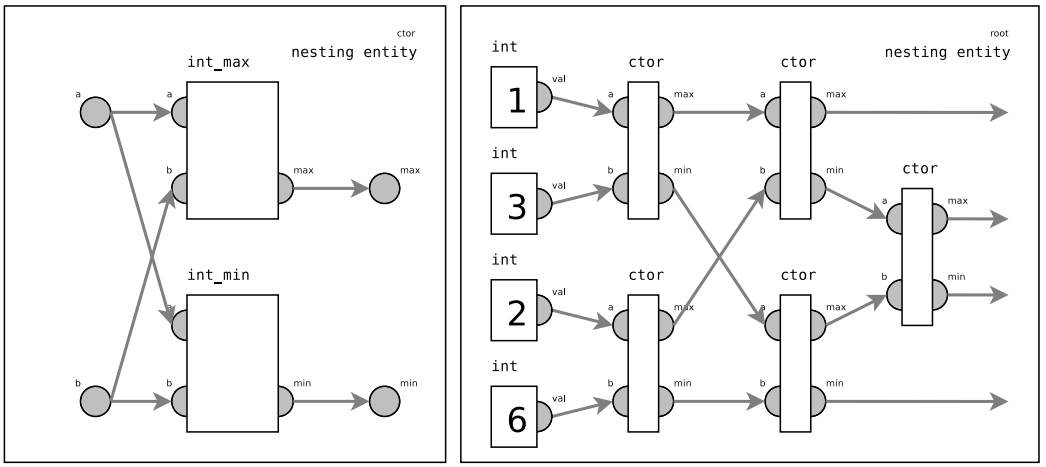


Figure 4.1: Model of a comparison network that sorts 4 integers. We would normally do something with the sorted numbers, but in this diagram the arrows on the right hand side are not connected to anything.

mation to other instances. Within the comparator entity, the oracles, advice and instances are linked together with dependency links.

Use of a nesting entity saves us some work in managing the model. Not only can we avoid instantiating a pair of instances for each comparator, but we can change the comparator entity in a subtle way, for example adding an advice that exports a boolean value describing whether or not the pair of oracles had to be swapped. Then this advice would immediately be available in all the instances.

4.2 Theory

The construal is therefore divided into a nonempty set of entities. The most trivial construal will simply have one entity, the root entity. More complex construals will contain different entities modelling different aspects of the situation, and these will be instantiated in the root instance, and each other. These entities each contain some oracles, advice, instances, and dependency links. The model is based at the root instance, and is defined to be a hierar-

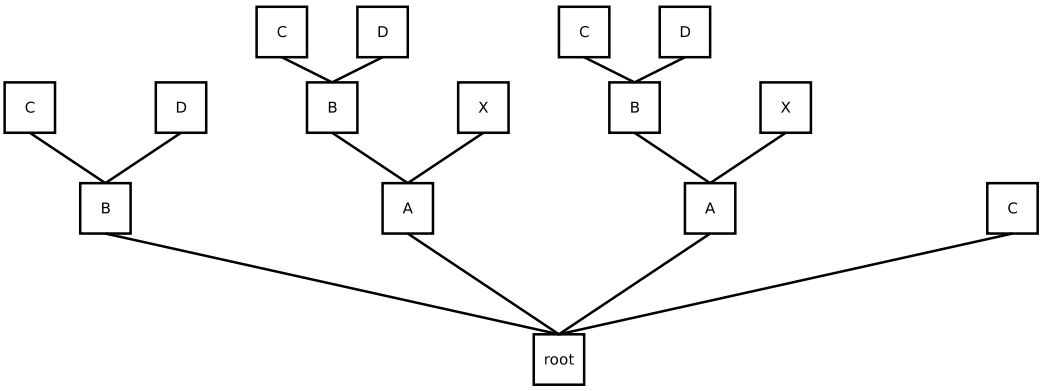


Figure 4.2: The instance tree. A, B, C, D, X and root are entities in the model. Each box is an instance of one of these entities.

chy of nesting instances. We explicitly forbid the instantiating of an entity within itself, either directly, or indirectly through the medium of another entity. Figure 4.2, figure 4.3 and figure 4.4 illustrate the structure of and relationships between instances and entities in an Asylum construal with six entities, and a complex nesting of instances.

Predefined entities are written in C code, they implement such features as addition, subtraction, or indeed any computation on data. In general, they can be thought of as accessing native, or native library functionality, and can thus be used to import any operating system features into the Asylum. As well as side effects, predefined entities can also have a form of animation state of their own, indeed this is a requirement if we are dealing with file access and other persistent data. Predefined entities should also be a source of agency. For example a predefined entity could represent the joystick, and provide two advice observables, one for the X axis and one for the Y axis.

Nesting entities are defined in terms of other instances and dependency. They can provide structure to a model, allowing the modeller to divide their construal into sections. This is possible because of the instances are encapsulated inside the nesting instance, in an object orientated style. This encapsulation means that the inside of an entity can have a different name space. This solves one of the problems Eden has suffered for some time.

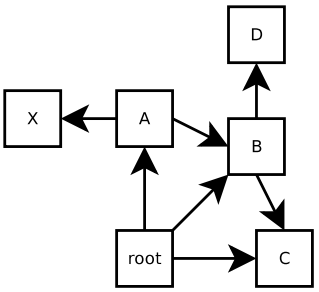


Figure 4.3: This shows the relationship between the entities. If an entity is instantiated inside another entity, there is a arrow from the outer entity to the inner entity. Cycles are explicitly forbidden in this graph. Note that root indirectly contains everything, and thus nothing can contain root.

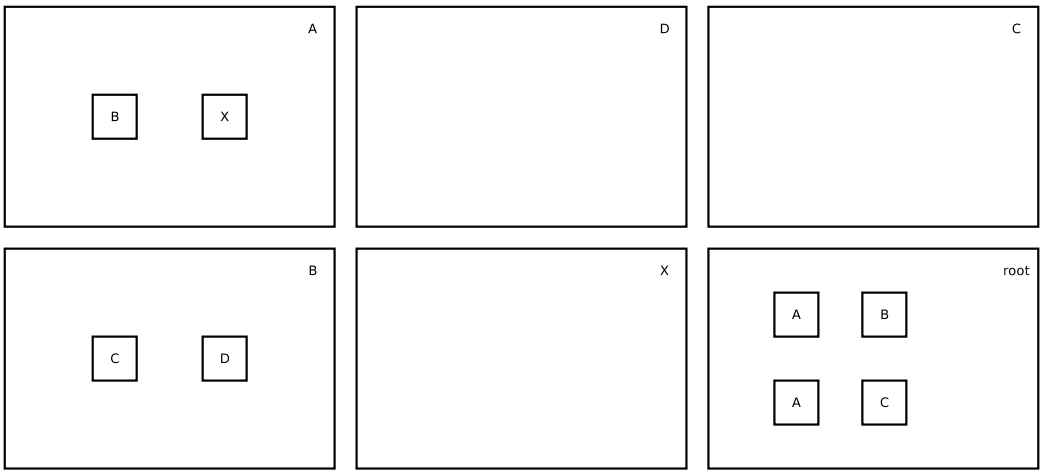


Figure 4.4: This shows the design of the six nesting entities in the model, and shows how the construal is partitioned into the six entities.

Conventionally, when programming, we use re-usable components, or libraries, in our code. This helps us to implement something complex without having to start from first principles each time. In Eden this is not really possible because you cannot import a model alongside your model. Because there is a single flat namespace in Eden, definitions from the library could transparently overwrite the definitions in your model, and this would subtly destroy the construal. Because the Asylum's nesting instances have their own namespace, it is conceivable that libraries of entities could be constructed, that represent different people's construals of different problems.

If someone implements a complex GUI widget, for example, they can package it into an entity and make it available for other people to simply import into their model and satisfy the oracle dependencies. In order for Empirical Modelling tools to become more advanced, more expressive, and more targeted for specific application domains (E.G. Primary school teaching), this functionality is a must have. For very large models (and the symbol table is already huge in Eden), being able to divide into a tree structure will make models much more manageable. Also, suppose the tool was used in a signal processing, engineering context, the tool could have a library of analogue electronic components that could be used for making construals in this domain.

We can also say something about the way that Human beings perceive their environment in a structured way. Typically we do not build construals with a flat structure of observables and dependency, but subdivide and divide into increasingly simpler and more literal construals until we are dealing with fundamental physical principles. For example, we think of a car as having a few functions, such as driving, braking, and needing petrol. If we are required to elaborate on this, we can think of it in terms of mechanics and chemistry. Typically, we will avoid dealing with the low-level details of a referent unless something about it is not consistent with our top-level construal. For example, the car is simply something that we turn on and use, until it breaks. In this case we would look under the bonnet to find more information.

4.3 Implementation

To be useful, any implementation of nesting entities must have the ability to maintain the implicit dependency between entity and instances. In object orientated programming, we typically are dealing with compiled code, and if a change is made to a class, the re-compile and re-execution takes care of mapping this change onto the instances. In an Empirical Modelling tool, we do not have the luxury of being about to assume that the entity will not change while instances exist. In general, all modifications to the model will be in one of the entities. When a modification to an entity is made, the tool must ensure that this change is repeated across all instances alive in the model.

Trivially, we can see that with an adjustment in naming discipline, any model built using a nesting entity framework can be re-implemented with a flat structure. All we need to do is "flatten" the tree one level at a time. This shows that an implementation of nesting entities is a form of sugaring, wrapping around an implementation of flat entities. This is evident in the ADM, which has a single definition store containing all the observables.

Because of this underlying flatness, it is not too difficult to implement checking for cyclic dependency or race conditions, we just treat everything as a flat surface of predefined instances and proceed as normal. There is one snag however: We need to check the entity as well as all its instances, because there is the possibility that the entity has not yet been instantiated, and thus will have no presence on the flat surface. In the asylum, this is implemented by using a reference instance for each entity, which is not strictly part of the model, but still falls into the path of the checking algorithms.

It is of interest that Eden already has a form of this idea, called virtual agency. This is like what has been described here, except that there is no dependency maintained between the groups, and some definition from which they are derived. I.e there is no dependency between entity and instance, within the virtual agency framework. Virtual agency is just a way of implementing a name-space within Eden, and usability problems with the syntax make use of this feature less worthwhile than it deserves.

Chapter 5

Type

5.1 Definition of "Type" and "Type Safety"

Type is a very well known, but abstract concept. In order to justify its inclusion in an Empirical Modelling tool, we have to make clear what it is we mean by "type". The definition follows from the knowledge that our observables are information, and hence contain values. If an observable is my name, it's value would be "David", my age in years would be 20. It is possible to derive subsets from the set of all possible values, for instance the set of all integers, or the set of all strings, and those sets are *types*.

Why do we might want to do this? When information is represented on a computer as data, it is much easier to define computational operations that work on specific sets of values. For example: there is no obvious meaning to the application of the subtraction operation to two strings. We avoid this problem by saying subtraction is only defined for numerical types (e.g. the set of values that form a group under "-"). It follows that certain dependencies cannot be made between certain observables, and that an EM tool must enforce these restrictions. To do this, the Asylum must know what set of values an instance can receive from its oracle, what possible values will be emitted as advice, and what it takes for types to be consistent.

We could substitute the undefined value "@", if an oracle provides an instance with a bad value, but this would cause usability problems, since in

almost every case, this situation will result from a mistake by the modeller, and an undefined value will not be correct within their construal. It is better to reject this dependency and give the modeller a message about the clash.

5.2 Type In Existing Tools

Let us review the type system present in the ADM implementation 'am' and in the EM tool Eden. Types are never an issue in 'am', because every datum is an integer. This greatly simplifies the interpreter since it can avoid this entire topic, but it reduces the expressiveness of the language, and thus makes forming a construal more difficult, if impossible.

Eden, on the other hand, has many different types, including "float" ,"int", "string", and "list". In many cases operators only apply to certain cases and eden will prevent the modeller from making definitions that cause type "clashes" for example:

```
1|> c is a + b;
2|> a = [1];
3|> b = 3;
/dcs/emp/empublic/linux-i686/bin/ttyeden-1.46: error: type clash: number type
required (got list) while executing file stdin near line 3, char 6:
b = 3;
```

From the first definition, we know that c, a and b must be the same numeric type, since that is the type to which the operation "+" applies. From the second definition, we know that a is a list. Since we know that "+" cannot be applied to a list, an error should be presented at this point. Unfortunately the type checking procedure does not seem to be active until all the dependencies in the definition of a are met, so we get a misleading error message when we try to (correctly) define b=3. Eden's problem is that it is checking the types of the operands when it performs an evaluation, rather than checking the types when the definition is made. It turns out that because of its design, Eden has no option but to do type checking this way. This is not an Empirical Modelling problem, it does not affect the modeller's ability to build a construal, neither does it make the tool unstable, but it *is* a usability problem.

Where Eden does succeed is in making type checking very implicit. Sometimes the term "manifest typing" is used to describe a language where the programmer defines the type of the data before using it, therefore making the compiler's type checking job very simple. (Another term for this is 'ascription'). We do not want to do that in Empirical Modelling since we want the modeller to build a model that expresses their construal, in terms of dependency, observation and agency. Type is an abstract notion that really has nothing to do with artifacts or referents.

Clearly Eden does not meet the requirement that needs the modeller to be informed as soon as possible when their construal is broken. There is a term "dynamic typing" used to describe type checking that is done at "run time". We have to be careful with this definition, but even in Empirical Modelling tools, where we have concurrent definition and evaluation going on, the evaluation can be thought of as "run time" and the definition as "compile time". It is thus possible to say:

> Eden is dynamically typed because it performs its type checking procedure when definitive expressions are evaluated, not when definitions are made.

What we really need is a system analogous to "static typing", or "compile-time type checking" where the problems are spotted when the modeller makes the definition. This is *not* possible in Eden for two reasons:

- Often models are built where redefinitions occur automatically, rather than under the modeller's supervision, to implement animation, or systems that react in some way that cannot be modelled without giving the system a definitive state that changes by itself. An alternative approach is suggested in Chapter 3.

- Eden has a list data-type that is a hybrid of C's array and struct data-types. Not only can its members be accessed with an integer index, but they do not have to all be the same type. That means that the type of the result of a function expression will vary depending on the run-time

value of its parameters, which makes static type checking impossible. What is the type of `a[i]` in this example?

```
1|> a is [1,2,3,4,5,6,"7"];
2|> b is a[i] + 3;
3|> i = 1;
4|> while(1) { i++; } /* A relatively innocent operation. */
/dcs/emp/empublic/linux-i686/bin/ttyeden-1.46: error: type clash: number type
required (got string) while executing file stdin near line 4, char 17:
```

5.3 Type In The Asylum

In the Asylum I have implemented a system where a type is inferred from the dependencies placed. It makes the system totally type safe without requiring the modeller to explicitly define what types observables are. The available types are:

- *integer* For discrete numbers, like counting sheep or selecting from a list. Equivalent to the platform's C "long" representation. Ideally we would want an arbitrary precision integer for this job, since integer overflow is unlikely to be part of every construal. Also, since different platforms have different representations of the same C type, networking different platforms (in the style of dtkeden) would not be easy. For example, if we are transferring the value of an integer across a network to a different host, running a different instance of the modelling environment, the two hosts might have differently *sized* integer types. This means overflow could occur over the network medium, and the two models would not correctly share the same information. The best we can do is agree on a standard representation, e.g. 32 bit twos complement, and have the code at both ends tuned to provide this for the specific platform. (I imagine this is how Eden does it).

- *real* For continuous data such as distances and velocities. Equivalent to the platform's C "float" representation. Again, an arbitrary precision real number representation might be better.

- *string* For incorporating text into a model. A UTF-8 encoded sequence of Unicode characters. This means that the string can contain any character from any language, mathematical symbol, or presentation form. So for example Korean or Arabic text can sit neatly inside Asylum observable data of type *string*.

- $array(\tau)$ Where τ is another type. This type is for dealing with a collection of identically typed objects of arbitrary size, for example a school of fish. The important function of arrays is that while the model is animating, the number of elements within the array can change, and that this type encompasses all the possible lengths of arrays. This is why all the types have to be the same, so that the type of an arbitrary member can be determined statically.

- $struct(\tau_0, \tau_1, \tau_2, ..., \tau_n)$ Where each τ can be a different type. This allows a 'record' of information, several observables to be grouped together in one structure. This could be useful for holding physical properties of an object, or for implementing a vector, or even a matrix. Since the types inside the structure are static, in the same way that a number's type is static, there is no problem with static type checking.

Structs and arrays are equivalent to the struct and array types in C and the tuple and list types found in functional languages. They separate the two functions present in the Eden 'list' type: that of having a structured type composed of lots of different observables, and that of needing to deal with arbitrary sized collections of similar observables. Unfortunately although the back-end Asylum code has support for arrays, this has not been brought forward to the hive (because I ran out of time), and there is very little support for structs even in the back-end. There is no reason why it could not still be implemented, however, and I did have these types in mind when I developed this design.

There are always operations that do not act on a specific type, or overloaded operators that can act on a range of types. For example, in Eden, the

"+" operator can add both floats and integers. The "?:" operator (declarative "if" statement) can apply to any type at all, E.G.

"`a is 1?"hello":"world";`"

or

"`a is 1?1.5:3.5`".

This is because some operations make use of more general protocols that can apply to a very wide range of data values. For example the action of copying a value, used in the "?:" operation, can be applied to any value of any type. This is often called "Polymorphism", which in general describes the application of some single protocol to a wide range of types.

To express the idea of 'any type will fit' notion, we can define a type hierarchy with a root "wildcard" type, which can hold any value, and to which generic protocols can be applied. This is inherited by all the other types, each of which defines additional protocols that can be applied to the data, and represents a subset of the set of values. When visualised in this 'tree' structure (this is not strictly a tree since some types inherit from multiple types), the types within the Asylum appear as in Figure 5.1. Clearly because of the infinite number of struct data types, the set of all types is unbounded. In general, we tend to think of values as having the most concrete possible types in the tree. This is because we try and be as specific as possible when describing the type of a value, since this allows us more freedom to apply whatever protocols we want. So we call [1,2,3] an array of integers, but it is also an array containing any type, and it is a wildcard type. If one type inherits another:

- It is a child of the type in figure 5.1.
- It can represent a subset of the values that the parent type can represent.
- Upon it can be applied any of the protocols that can be applied to the parent type.
- There may exist some protocols that can be applied to the child, but not the parent.

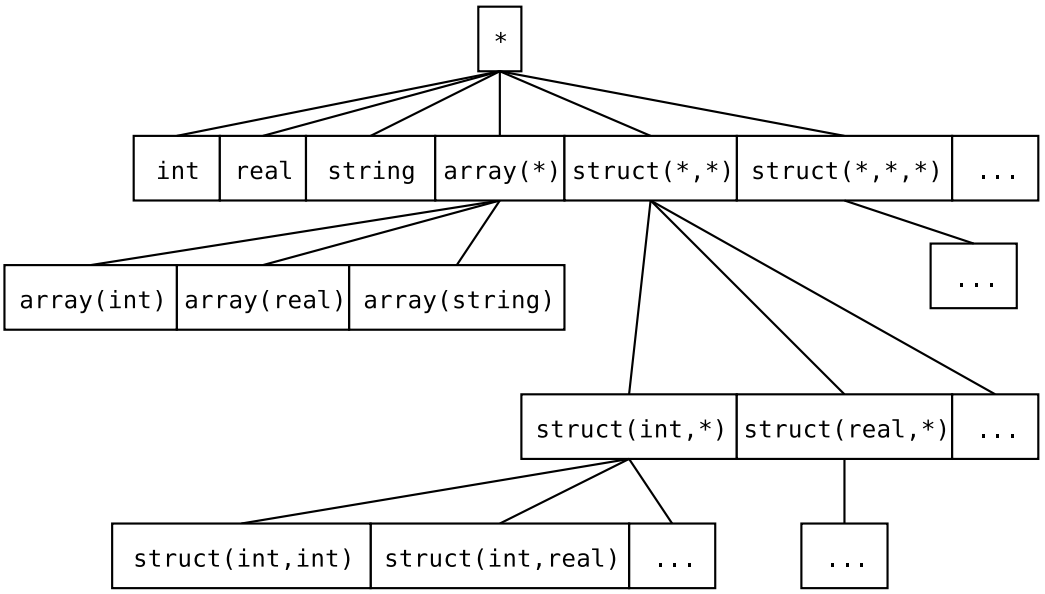


Figure 5.1: The hierarchy of possible types in the Asylum design

The oracles, and advice of any instance in the Asylum, will each have an active type from the tree. For example in the predefined entity `arr_lookup`, is provided the capability to extract a member from an array. One oracle provides the array, another provides the index, and there is a single advice that provides the specified member data, see Figure 5.2. Clearly the index should be of integer type, and because we're looking up the member of an array, the array should of type *array*(*something*), in fact an array of anything. I have used the compact notation `int` for integer type, `[`τ`]`, for an array of some type, $\{\tau_0, \tau_1, \tau_2, ..., \tau_n\}$ for structures and `*` for wildcard type. This means an 'array of anything' has type `[*]`.

These types represent restrictions placed on the data, and these restrictions stem from the nature of the entity's programmed behaviour, and thus are fundamental. For that reason they are called the "principal" types. This terminology comes from the Hindley-Milner type inference system used in many functional languages. As I have come to understand, there is an almost complete intersection between type theory in functional languages and in what I wanted to implement in the Asylum.

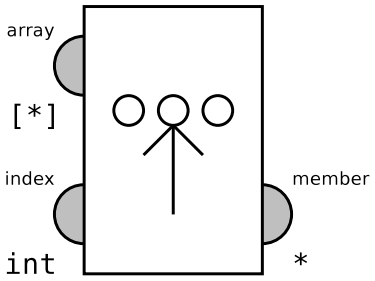


Figure 5.2: The oracle and advice of the `arr_lookup` predefined entity.

Now we understand what types different oracles and advice provide, the rest of the problem is in finding out if an oracle and an advice can be connected. Trivially they are compatible if both types are the same. But what about when we want to use the `arr_lookup` entity with an array of integers? Clearly we can look up a member of an array of integers just as well as we can look up a member from any other type of array so this is safe.

When a dependency is made by linking an oracle to an advice, the value held by those two observables is *identical*. Since the set of possible values is the same at both ends, the type is the same at both ends. Before the link is made, the ends each have their own types, and therefore support different sets of values. When the link is made, we have to take the intersection of the types to ensure that values are acceptable at both ends. This means the types might become more constrained.

The types of oracles and advice actually change when a link is made, so an oracle might have an active type which is more restrictive than its principal type. In this case the oracle changes from `[*]` to `[int]`. This forces the advice to change type to `int` as well, but for now we shall concentrate on the intersection of the types `[*]` and `[int]`.

In general, we are looking to take two types, τ_0 and τ_1, and producing a third type τ_2 that will replace both. This type might not necessarily be a copy of either, for instance the structure {`int,*`} and the structure {`*,int`} intersect to give {`int,int`}.

τ_0	τ_1	Unify(τ_0,τ_1) =
*	*	*
a	*	a
a	a	a
a	b	Unify(b,a)
real	int	reject dependency!
[a]	[b]	[Unify(a,b)]
$\{a_0,\ldots,a_n\}$	$\{b_0,\ldots,b_n\}$	$\{$Unify(a_0,b_0),...,Unify(a_n,b_n)$\}$

Table 5.1: Interesting properties of the function that attempts to unify the types at either end of a dependency.

The type τ_2 is a child of both τ_0 and τ_1, and thus inherits both of their capabilities, and can represent the intersection of their values. We have no need to make the new type any more restrictive than it needs to be, so we choose the least specific type of the possible candidates. Implementing this algorithm is almost trivial with the Asylum data-types. This process is called unification of types, and is represented in the code by the function `asylum_type_weakestmatch()`. Table 5.1 shows some of the results of this function.

This only leaves the matter of how the types of oracles and advice of an instance can be interdependent. This is necessary because otherwise operations with more than one operand cannot be supported. In the Asylum, instances receive messages when their type changes, and they act on this by maybe changing some of their other oracles/advice to match, and propagating the change. For example on hearing that it's oracle had changed type from `[*]` to `[int]`, an `arr_lookup` instance would change its advice's type to `int` and notify any dependents of that advice.

This process is an explicit part of a predefined instance's programmed behaviour. In conclusion these type conditions must be met at all times in the Asylum:

- Type at either end of a dependency must be the same.

- There may be type interdependencies between instances that must be maintained.
- Every advice and oracle of an entity has a principal type that arises from its defined behaviour.
- Every advice and oracle of an instance has an active type that arises from its entity's principal type, and any dependencies in place with other instances.
- The acting type of a nesting entity's oracle or advice must be the same as any advice or oracle it is connected to, both instances nested within the entity, and instances that instances of the nesting entity are linked to.

If these conditions are met, we can avoid the overheard of dynamic type checking, and check for type safety when dependencies are placed.

5.4 Comparison To Type In Functional Programming

Take the following example, written in SML:

```
- fun arr_lookup (x::[]) n = x
    | arr_lookup (x::xs) 0 = x
    | arr_lookup (x::xs) n = arr_lookup xs (n-1);
> val 'a arr_lookup = fn : 'a list -> int -> 'a
```

The types of the function (compare to entity) have been inferred from the recursive definition, as a "'a list" (`[*]`), an int (`int`), and "'a" whatever type was inside the list (`*`). In the next bit of code, we apply the function in the context of looking up an index of a list of reals (compare to instantiating the entity and connecting its oracle to a list of reals).

```
- fun f n = arr_lookup [1.0,2.0,3.0,10.0,2.0] n;
> val f = fn : int -> real
```

The SML interpreter unifies the type `'a list` with `real list`, and thus the value returned by the function (the advice) is constrained to a real as well.

5.5 The Implementation

There is no one area of the Asylum's code where type is programmed. Rather there are a group of functions that are useful for working with types, and the actual type code is distributed around. Each predefined entity defines the principal type of its advice and oracles, and this type is the initial acting type of new instances. When a new oracle or advice is added to a nesting entity, it has a principal type of `[*]`. We can think of this as the base case in an inductive proof that the Asylum is type safe. See figure 5.3 for an illustration of a model with no dependency and all active types set to the principal type. The types are displayed in pairs, e.g. `*,*`. The first type is the principal type, the second, the active type.

As dependencies are added to the model, the types will change in the manner described above. See figure 5.4, the same model with dependencies linking everything together. It is trivial to see how types spread through the Asylum. When a new link is placed, a message is sent to the instances at both ends. They each re-evaluate the acting types of all their oracles and advice, and send messages on to their dependent or depending instances. If the type does not need changing, no more messages are propagated. A wave of messages is passed from instance to instance until all the types settle into a consistent pattern.

The code that does the actual re-evaluation of acting types, and the propagation of messages, is held within each predefined instance. This allows each instance to have different principal types and different interdependencies between its oracles and advice. It would be better for the type propagation algorithm to be factored out and generalised into a library function, where it can be considered bug free, and optimised. This has not been done, however, and at this time all predefined instances have to contain code to propagate

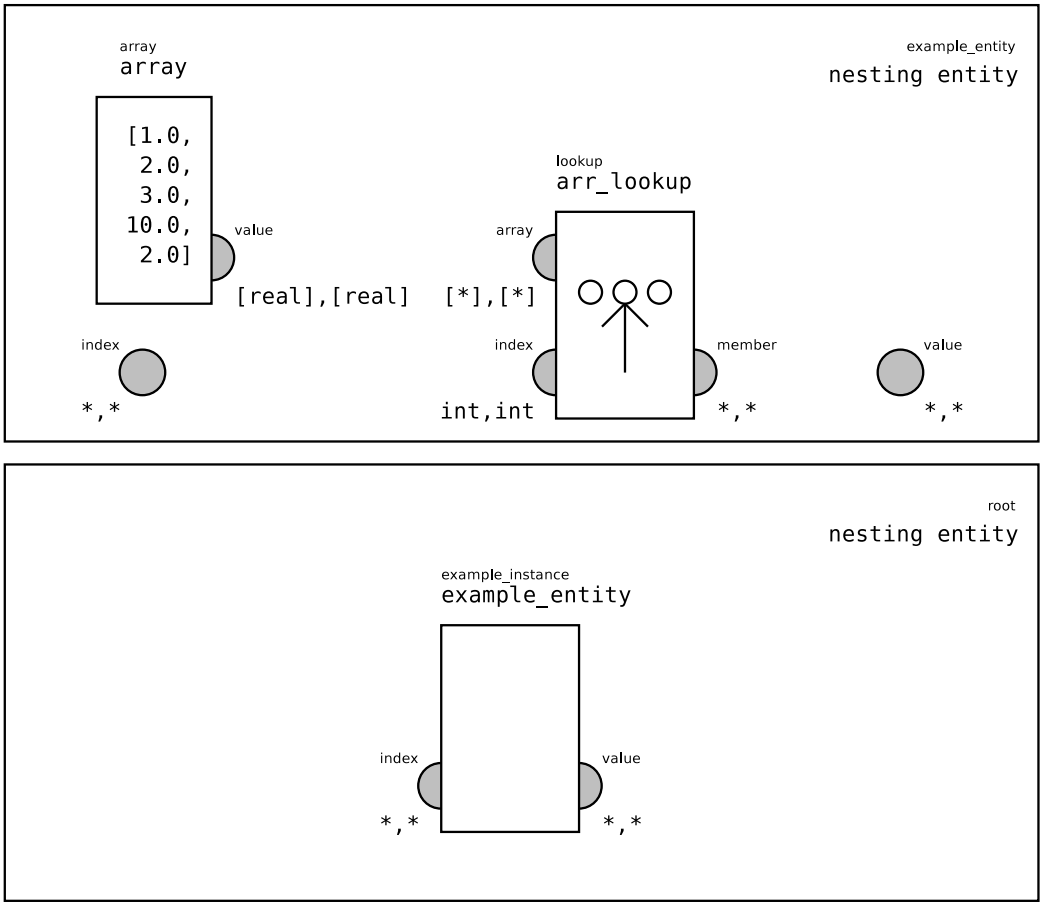


Figure 5.3: A demonstration of principle types in a model with no dependency. Note that `array` is a fictional predefined entity that simply provides an array of reals.

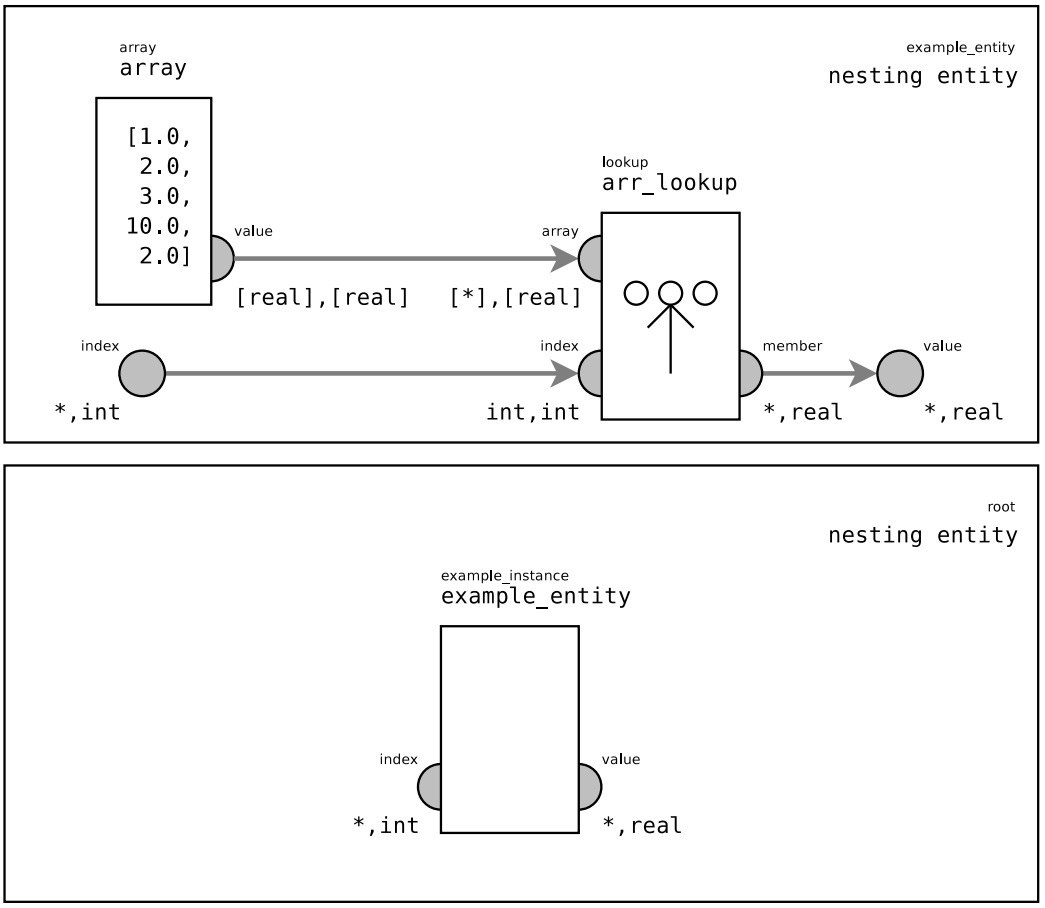


Figure 5.4: A demonstration of active types in a model, arising from dependency.

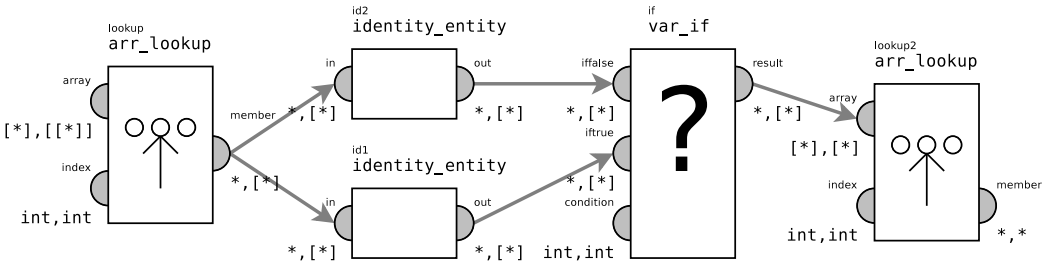


Figure 5.5: A more complicated model.

messages to their dependent neighbours.

It is less trivial to see how the types should be released when a link is removed. Clearly we need to re-evaluate the type at both sides of the deleted link, but how do we know what the new type should be? It must be somewhere between the principal type and the old type, but it is difficult to know the extent of influence that the deleted link had on each instance.

To clarify this problem, the act of deleting a link, is equivalent (in terms of the resultant observable types) as removing every link, then relinking everything *except* the link that we wanted to remove. The problem can be simplified into a graph problem: Observe the model in figure 5.5. This model uses two new entities - the `identity_entity` simply copies its oracle onto its advice, and the `var_if` entity chooses between its oracles `iftrue` and `iffalse` depending on whether its oracle `condition` is 0(false) or some other number(true). In Eden, we might represent this model as so:

```
lookup_member is lookup_array[lookup_index];
id1_in is lookup_member;
id1_out is id1_in;
id2_in is lookup_member;
id2_out is id2_in;
if_iftrue is id1_out;
if_iffalse is id2_out;
if_result is if_condition ? if_iftrue : if_iffalse;
lookup2_member is lookup2_array[lookup2_index];
lookup2_array is if_result;
```

It takes a while to understand, but `lookup_array` must contain a list of lists (or an array of arrays in the Asylum). This is a demonstration of

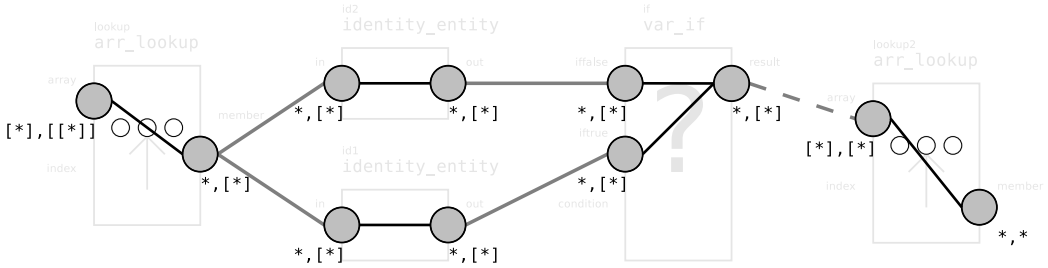


Figure 5.6: The same model as figure 5.5, simplified.

how although the flow of information in a dependency network is an acyclic directed graph, the flow of type is not directed, and thus cycles can arise. In this model, the type flows from the "end" of the model to the "beginning", from the instance called "lookup2".

The following code demonstrates what happens in Eden if we fail to use a list of the correct type on the far left of this model (note we have to fill in all the undefined oracles before the type error is shown):

```
11|> myarray is [1,2,3,4,5];
12|> myindex is 1;
14|> if_condition is 1;
15|> lookup_index is myindex;
13|> lookup2_index is myindex;
16|> lookup_array is myarray;
/dcs/emp/empublic/linux-i686/bin/ttyeden-1.46: error: index error: list or
string required (got int, when trying to find 1th item) while executing file
stdin near line 16, char 24:
lookup_array is myarray;
```

This error complains that the members of the list `myarray` are integers, rather than lists, (or strings which can also be used with the `a[i]` syntax).

When we simplify this model into a graph of connected nodes, we get figure 5.6. So we can see that the principal type of the observable lookup2_array has spread throughout the model. Now if we delete the link ending at the `array` oracle belonging to the `lookup2` instance (this link is dashed in figure 5.6), we must change the active types of many of the observables in the model. An informal study of this specific example shows us that the types

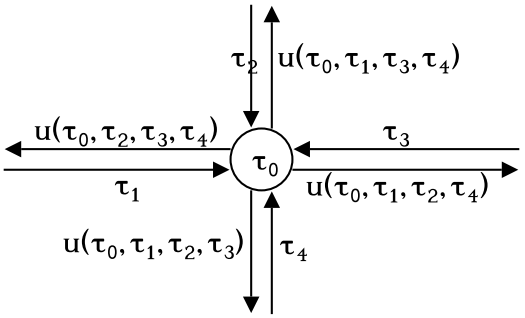


Figure 5.7: A node in the Asylum's type release algorithm. The function u(...) means "The unification of a set of types"

to the left of the deleted link should all decay to their principal types, but what kind of algorithm could decide this in the general case?

5.6 A Difficult Problem

The algorithm present in the Asylum is not quite correct, it fails to relax the types fully when there is a cycle in the graph. You can work around this by breaking and relinking the cycle, but this is not really ideal from a usability perspective. The algorithm works by recording the types that flow into the nodes, from all directions, and using this information to project the types out of the node in the other directions. If a node is connected to 4 other nodes (and hence has 4 edges connected to it), there will be 4 types recorded coming into that node, and 4 types will be generated leaving the node. For an arbitrary edge attached to a node, the type flowing out of that node is the unification of all the types coming into the node, *except* the type coming in along the same edge. See figure 5.7.

I felt it would be beneficial to store a small amount of localised information to help the algorithm make its decision with minimal computational expense. By recording what edge the type came from, it seemed that we would know to back down if that edge was removed. Unfortunately this approach assumes that the type which is being projected out, along an edge,

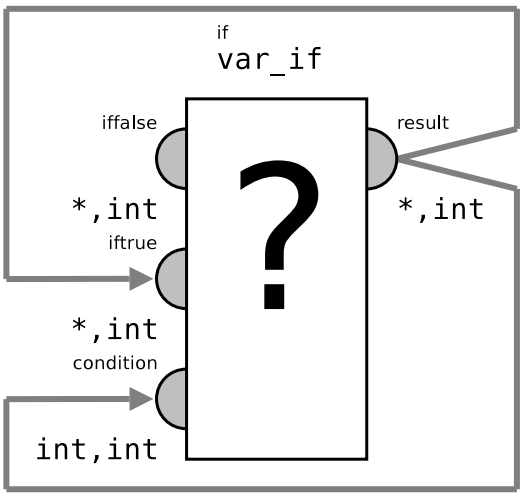


Figure 5.8: A simple model where the type back tracking mechanism fails.

will never find its way back to the node down another edge. This assumption breaks down in the case of a cyclic graph. The simplest model that demonstrates this failing is a single `var_if` instance, linked in a circle (this is possible with evaluation links). The type of the result should decay back to `*` after the link to the `condition` oracle is removed (this model is graphically represented in figure 5.8) Unfortunately it remains as `int` type. Here is a transcript of a session in the "shiver" shell that demonstrates this "feature":

```
shiver$ load lib/plugins/libstd.so var_if 3 1 condition iftrue iffalse result
Done.
shiver$ addinst root var_if if
Adding an instance of var_if in root called if... Done.
shiver$ elink root if result if iftrue
Done.
shiver$ elink root if result if condition
Done.
shiver$ inspect /if/result
Observable: 0x805f23c has type integer and is compromised.
  @
shiver$ unlink root if condition
Done.
shiver$ inspect /if/result
Observable: 0x805f23c has type integer and is compromised.
  @
//$ SHOULD BE "has type * and is compromised."
```

It is interesting to note that conventional functional programming languages have not run into this problem at all. The reason is that declarative languages typically do not support this interactive mode of operation, and removing the application of a function is impossible. Even in interpreters like mosml and GHCi, it is not possible, the closest equivalent operation is to create a new definition with the same name. This of course simply infers the types from scratch all over again.

I have concluded that there is no trivial algorithm for solving this problem. Interestingly, the problem itself is almost identical to the problem of generalised garbage collection - that of knowing if part of a graph (in this case used to model a data structure) is totally isolated from a given root node. My current solution is like that of reference counting for garbage collection, it is flawed in the case of cyclic graphs. Solutions to the garbage collection problem (such as mark and sweep) are specialised for that problem. If there is a solution to the back-typing problem, it would most likely take advantage of the specific character of the problem, rather than utilising some general technique that would work for an arbitrary graph.

It appears that the only ways to determine what to do in this situation, are computationally intensive (e.g. re-inferring the active type for every node when an edge is removed) or memory intensive (e.g. recording all the indirect

links to each node, at each node). This may sound bad, but in reality it is only a factor to be added to the task of detecting cyclic dependencies and race conditions, which is also intensive. The business of type checking a dynamic computational system, like so many other aspects of an Empirical Modelling tool, seems to be very expensive in terms of system resources.

Having said this, an Empirical Modelling tool absolutely requires a static type checking algorithm. There are mistakes that the modeller can make, and the quicker he is informed of such problems, the easier it will be for him to build a construal. Like other computationally intense algorithms that must be used, this is simply another complexity to consider when creating an Empirical Modelling tool.

Chapter 6

Development of the Asylum

This chapter is concerned with the details of implementing the Asylum dependency maintenance library in C. This includes design decisions made, software engineering techniques used, and the algorithms and ideas used to implement the various interesting features of the Asylum. At the end of this chapter is a section to explain how to use the Asylum source package, and run the "Shiver" Asylum shell interpreter.

6.1 Design

6.1.1 Structure

The Asylum package has a tier architecture: At the lowest level we are dealing with the support library Glib, and various native features. Directly above this is a dependency maintainer (often referred to as "The Asylum") and a set of predefined instances that can be plugged into the dependency maintainer. Above that is a symbol table, which effectively mirrors the Asylum's API, but provides names for the various beings within the Asylum (instead of numbers or pointers) and catches some errors. This API is called "The Hive" and is useful for implementing friendly user interfaces (such as Shiver). The Hive also has the ability to load plug-in library files from disk, and extract predefined entity definitions from these library files, for use in the internal Asylum.

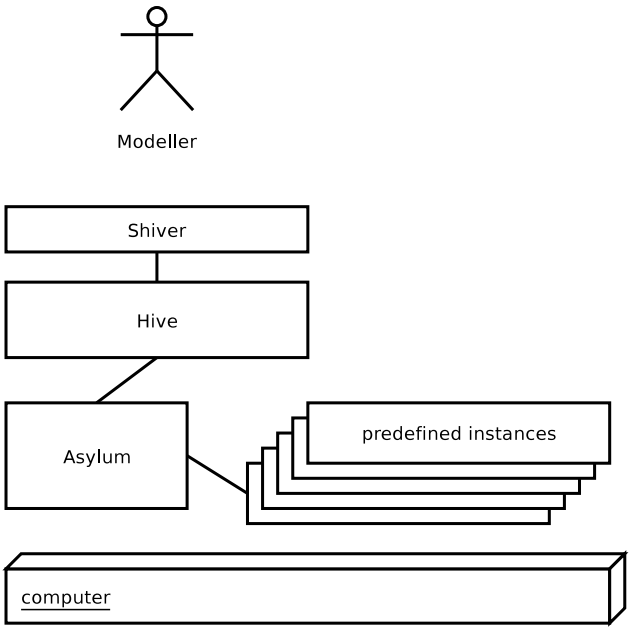


Figure 6.1: Top level overview of the Asylum project.

On top of the Hive lies the only user interface that exists at the moment, "Shiver". This is just a shell that provides a command line interface to calling the Hive's functions. Figure 6.1 illustrates the design of the project. Ironically, because Shiver is a tool that exists to demonstrate the features and power of the Asylum, rather than being a useful Empirical Modelling tool, it suffers from many of the problems that "am" faces. It is probably too much work to implement both the back-end and the front-end of an Empirical Modelling tool. Hopefully, since the back-end of the Asylum is practically complete and well documented, people can use it in future. On the other hand, people often want a specific maintainer, and interfacing with someone else's arbitrary code can be more work than re-writing it yourself.

The advantage of dividing the project up into these parts (apart from the ability to test the different components individually), is that certain applications of the Asylum could use the dependency maintainer in an embedded way. The idea was someone could develop a model using the interactive front end, save it, and then embed it into a program, E.G. a game. The game could

link with the Asylum API, and use the dependency maintainer without the overhead of the symbol table or the user interface getting in the way.

6.1.2 Early Decisions

As mentioned in section 2.5, the Asylum's dependency maintenance back-end was designed to present a set of very short interactions to the modeller. Rather than having to provide a complete entity, the Asylum is designed to allow an Entity to be built up in small sections, and be instantiated and modified at the same time.

So in contrast with traditional tools that parse a string of input, and make quite a large change to the observables and dependencies being represented within, the Asylum has a large number of functions that can be called to make small changes. These functions include things like putting an instance into a nesting entity, or adding an oracle to a nesting entity. Each function has an associated "undo" function that performs a similar function to building the model from scratch without that feature present (although obviously it does not do it this way). This means the number of possible sequences of interactions that will result in a given model, is extremely vast. This is what we want, because we are empowering the modeller with choice, and room to experiment. We also need to forgive the modeller for mistakes, so that a real process of trial and improvement can develop.

It is quite tricky to implement this dynamic strategy for the following reasons:

- We have no control over the order in which interactions occur. We have to deal with the undefined behaviour of an instance without oracles connected. We have to propagate changes throughout the network when changes are made. We have to deal with changes made to entities both before and after instances are made.

- We have to deal with dependencies in the structure of the model, that is if an instance or oracle is deleted, any links to this oracle or advice

must also be deleted. This includes links from the oracle to the instance inside the entity, and links from the instances of this entity, to other instances, that lie alongside them. If an entity is deleted, all its instances, and any links to its instances, must be deleted as well.

On top of this, we have to check the types of oracles and advice before making any links. If an entity's internally unlinked oracle, is externally linked in two different instances, to two different types, we have to make sure that the internal link is to an instance that can support both of these types. In other words, we have to enforce the polymorph-ism of entities.

As mentioned before, implementing nesting instances can be thought of as a virtual layer on top of an implementation of predefined instances. We simply have to project the modifications of the nesting entity itself, onto all instances of the entity. The oracle and advice connections are just placeholders, that record what internal instances must be connected to external interfaces when a "virtual" connection is made. There are in fact 3 layers in the Asylum:

- The nesting entity layer is the point of contact with the modeller. Most modeller actions are contained within a nesting entity. For example we can add an instance or make a link within an entity. If we are "clocking" an instance, we clock a specific instance within an entity. This layer simply projects the modifications onto the nesting instances, if there are any, and keeps a record of the changes in a template file that is used to make new instances.

  Note that there will always be at least one instance of each nesting entity, the reference instance. As described in section 4.3, this is necessary to check an entity before it is instantiated. If we do this, we only need logic that checks for problems within the instances. As it turned out, this is not such a good idea because instances have side effects, and thus the reference instance is not transparent to the modeller. (It appears as a phantom instance that causes side effects that they have no control over). To correct this, we either need to scrap the reference

instance idea, or implement some mechanism for disabling side effects within the reference instance.

- The nesting instance layer takes the modification and projects it onto the "flat" predefined instance layer below. For some modifications, like the clocking of an instance, or the linking of two instances inside a nesting instance, this is trivial. Where modifications involve the advice or oracles of a nesting instance, it is more involved, since we must determine what predefined instances external to the instance are being affected by the modification.

- The predefined instance layer deals with joining together predefined instances in a manner which implements the behaviour of multiple nesting instances.

6.2 Tools and Methodologies

C is a flexible, portable language. There are many libraries that the Asylum could link to, that can extend its functionality. C also compiles natively, without the overhead of a virtual machine. This makes it very fast during run time. The use of Glib as a portability wrapper and as a source of abstract data structures such as re-sizable arrays, event loops, and hash-tables, meant the programming task was focused on the more important details of implementing the Asylum's algorithms.

Although C is not an object orientated language, it is possible to write object orientated software in it. As an analogy, C is not a safe language, as it is possible to write C code that has undefined behaviour, but it is possible to write safe software in C, if we are aware that we are not invoking this undefined behaviour. In a similar manner, C has support for structures and pointers, which can be used to implement the features of encapsulation, polymorph-ism and late binding that we would expect from an object orientated system. The nature of instances and entities is that there are two distinct types (predefined and nesting) that present the same interface, and

the fact that a large number of specific predefined instances were created with different functionalities adds to this. By inheriting from a common base type, we factor out common code and present a polymorphic interface to other functions, which allows the Asylum to be easily extended by adding in new predefined entities. This is a stylistic design decision, which makes the Asylum's code slightly more elegant.

To implement inheritance with C structures, we make the first element of the child structure an instance of the parent structure. Since the structures are aligned at the first byte, we can transparently treat the child structure as an instance of the parent structure. By using function pointers inside the parent structure, that are overridden by the child constructor, we can execute a specific function at run time, that depends on the specific instance we are dealing with. The calling function need not know what the instance is, it can just call the function.

C is not a safe language, so to make the Asylum programming task easier, a tool called Valgrind was used. Valgrind can run native C code in a sandbox, essentially an x86 emulator, which is programmed to spot bugs as early as possible, and hence make debugging much easier. Many bugs in a C program will simply cause a segmentation fault, or invoke some undefined behaviour which is very hard to trace. Valgrind gives the programmer the ability to spot these problems early on and deal with them. As well as array overruns and null pointer dereferences, Valgrind can keep track of allocated memory, and detect memory leaks. This is an invaluable tool when developing C programs, and every C programmer should use it.

Because of the nature of the development process in a language like C, it can be quite hard to have much involvement in the execution process. If the program does not quite behave as we expect, we must use a debugger to trace exactly what is going on. GDB is an excellent command line debugger that allows us to insert break points in our programming code, trace executions, and give stack traces at any stage. If the software is compiled with debugging symbols, we can step through a program a line at a time, as if it were an interpreted script. This is vital when debugging the behaviour of complex and intricate algorithms.

In the code itself, special macros were used to aid the debugging and development process. Three levels of `DEBUG` macros were used to display verbose messages relating to the processes occurring inside the software. This verbosity can be "compiled out" of the final software, after testing has occurred. A `FIXME` macro was used in places where code was yet to be written, or known bugs were yet to be fixed. The behaviour of this macro was to print a message in red text on the console, showing what the problem was, and the exact C file and line number where the `FIXME` was located.

Because the expected behaviour of the Asylum API was known before it was implemented, it was possible to design small tests that used this API. Extremely simple models, such as an ascending sequence of numbers, were constructed in little C programs that simply called the API functions to build and execute the model. In "extreme programming" style, these tests were used as a point of reference for determining how much work remained to be done. Once the "shiver" shell had been written around the Hive API, tests were no longer needed since it was possible to interact with the Asylum in a much more dynamic way.

As mentioned in section 2.5, the original "am" implementation was poorly documented. In order to avoid this problem in the Asylum, a tool called Doxygen was used. Doxygen produces HTML, man, and latex documentation automatically, from comments written in a special format in the source code. This is ideal, since the comments can be maintained along with the source code itself, making for more reliable documentation. This documentation is suitable for people making use of the Asylum as a library, and for people thinking of understanding the Asylum's code itself. The ratio of comments to actual code in the Asylum, is well over 2:1. The documentation has complete coverage and is reasonably detailed, especially when compared alongside other Empirical Modelling tools and open source projects that the author has experienced.

Flex and bison (previously lex and yacc) were not used in the shiver tool. This is because the interaction with the model is so fine-grained that a syntax is not needed. Instead, libreadline was used. Libreadline is a Unix library that allows interactive use of `stdin`. It supplies a command history, which

can be searched, and also tab completion. This is a much better solution to the problem, than the grammars used by "am", and has more in common with ttyeden.

6.3 Instance Representation

The use of natively compiled C code to implement predefined instances, gives us not only the ability to do computations such as addition and subtraction, but to interface with external libraries. It is possible for an instance to make use of, E.G. a cryptography library to encrypt data supplied through its oracles. A predefined instance could maintain, or perform queries on an SQL database. Predefined entities should also be able to be used as a source of agency, however this has not been implemented in the Asylum. The only source of agency is modeller input to change values and dependencies, and the clocking mechanism.

Because we have such tight control of the entity within the C code, we can implement interesting instances that do more than model a simple operator or function. A good example of this is the `var_default` entity, used to implement the initial animation state, when using a loop back evaluation dependency. This predefined entity has two oracles, and one advice. Under normal operation the observable data simply flows straight through, but if the data is undefined ("@"), it is substituted with the value on the other oracle. This allows us to break the loop of undefinition that is initially present when we make a loop back dependency.

It also has interesting applications when packaging behaviour into nesting entities. We can provide a default value for oracles that are not satisfied. This is actually implicitly used in Eden's donald and scout notations. If there is a broken dependency somewhere leading up to a graphical display, the graphical display simply draws a grey window instead. In general, we do not want the whole model to fall to pieces if there is one undefined observable, so occasionally we will want to use default values to keep the model usable while modifications are made. This is analogous to the use of "pull up resistors"

at the interfaces to circuit boards, in digital electronics.

Because there are a large number of entities that are very similar, E.G. entities that implement arithmetic operations, a method was devised for automatically generating code for these predefined entities, based on a much shorter description. This basic description would include the names and types (which must be concrete, i.e. these auto-generated entities are monomorphic rather than polymorphic) of the oracles and advice of the entity. The description also includes a small segment of code that is used to recompute the advice based on the oracles, when the oracles change value. Here is an example of a description of a monomorphic entity that computes the "$\leq$" operation. (Note the use of the DEBUG macro to print a debug message when the calculation is performed.)

```
ORACLE_TYPES='a=INTEGER=sizeof(glong) b=INTEGER=sizeof(glong)'
RESULT_TYPE='glong'
NEW_RESULT_TYPE='INTEGER'
ORUP_PREPARE_RESULT='inst->result_data = *(glong*)inst->a.obs.data <=
                                         *(glong*)inst->b.obs.data;
                     DEBUG3r(g_print("(%ld<=%ld)=%ld\n",
                                     *(glong*)inst->a.obs.data,
                                     *(glong*)inst->b.obs.data,
                                     inst->result_data  ));'
```

These mini descriptions are processed by a bash script, and merged into a "template" C file, to produce some auto-generated source code, which then compiles into the predefined entity.

Predefined instance constructors have an argument, which can be an arbitrary array of bytes, and can thus be used to transfer arbitrary information to the constructor of the instance. In general it was decided that arguments for nesting entities have no real value, since information can be imported into the entity through oracles. In the original ADM, arguments are required because all of the observables sit in the same namespace. Clearly this is no longer required. In some predefined entities, however, arguments are useful to augment the definition of the entity. For example in the definition of the "integer" predefined entity, that supplies a literal integer, we would normally require a different entity to represent each number. This is not

necessary however since we can supply the specific number as an argument to the instance's constructor.

This is not to be confused with the act of modifying the predefined instance's observable after it is instantiated. This is possible, but these changes are carried out in the context of a nesting instance, not a nesting entity. We modify the value of an observable in a specific nesting instance, whereas with the constructor argument, we modify the nature of the observable in all instances currently alive, and the instances yet to be made. The constructor argument can be thought of as supplying an initial, or default value, which can afterwards be changed by the modeller. Most predefined entity constructors, and all nesting entity constructors ignore this argument, however.

6.4 Agency

In a C program, our "agency" is always supplied by a file descriptor. This allows us to interact with the system, and hence IO peripherals such as mice and joysticks. It also allows us to use the network as a source of agency. In general we must poll this file descriptor to see if there is agency pending, that needs to be dealt with. This is possible using an event loop.

An event loop is a technique often used in GUI toolkits, where the program has to deal with multiple different sorts of events (or agency), such as mouse and keyboard interaction, timers, watching files stored on disk, and network activity. The principle is to keep a set of polled file descriptors open, and a set of functions assigned to deal with an event, if it is raised. If a mouse click occurs, a set of functions are called to deal with this mouse click. This allows us to concurrently deal with (and wait upon) multiple sources of agency without using threads.

In the Asylum, we could use an event loop to watch a number of sources of agency, including the modeller making modifications to the model, any graphical interface the model has, and the sort of user interface hardware used to make models more realistic (E.G. a steering wheel). If an event is detected, we can change the value of an observable, and propagate this

change to the observable's dependent instances.

Another form of agency is the activity of the modeller editing the value of the advice of certain instances. This causes the propagation of change through the model that we associate with agency. This activity is subject to certain restrictions, we cannot allow the concurrent updating of two observables that depend on each other, since this causes a race condition. This is the mechanism used to communicate values to the embedded model in the `9.gtkblocks.bin` program.

A third form of agency has already been the subject of much discussion in this report, and that is the clocking of instances to allow the animation of state over time. The modeller adds instances to a set of "clocked instances". When the clock tick occurs (when this happens is controlled by the modeller), all the clocked instances evaluate their oracles and update their dependents with new advice. Again we have to prevent race conditions caused by concurrently updating two dependent observables.

This concurrency of agency is implicit in clocked agency, but explicit in the editing observables activity, where each edit is a separate function call of the Asylum or Hive API. When we have made all the concurrent changes, we call a special function that commits the changes to the Asylum. This functionality is similar to the command "`autocalc`", in Eden, which temporarily disables the resolution of dependency while we make concurrent changes, and also the "`commit`" functionality within the Java dependency maintainers.

6.5 Observable Representation

In the asylum, observables are represented with an arbitrary length piece of data. Some observables, such as those of those of integer type, have a fixed length that never changes. Others, such as strings and arrays, have a variable length that can change during agency events. When dealing with arbitrary observable data types, in polymorphic predefined entities, we must ensure that we can deal with arbitrary sized data as well.

There is thus a special format for storing each observable data segment, and this format is determined by the type of the data. Integer and real data is simply copied into a sufficiently large block of allocated memory. On x86 platforms, this is 4 bytes, but this can vary on other platforms. String data is encoded (using the UTF-8 encoding) as a `NUL` terminated array of bytes, of arbitrary size.

Arrays of data have a more complex structure. Since the elements in the array have arbitrary size, despite all being the same type, we need to provide an index at the beginning of the array. The array is a block of memory, the first few bytes contain an integer representation of the number of elements in the array. After this, there is a number for each member of the array, this number is the number of bytes into the array, that each member is located. After this index, each member is represented back to back, the first part of each member block is the size of the member.

This has been implemented inside the Asylum's predefined entities that deal with arrays, "`arr_compose`", an array aggregate, "`arr_lookup`", which is used to access an arbitrary member of an array by its index, and the pair "`int_arr_sum`" and "`real_arr_sum`", which perform arithmetic on arrays. Writing platform independent code to deal with memory at this low level was quite difficult, since alignment issues had to be taken into account.

The struct representation was never attempted inside the Asylum, but should be much more simple than the array representation.

Of interest, is the manner in which the array aggregate "`arr_compose`" predefined instance functions. The purpose of this instance is to take a collection of identically typed observables, from various instances, and collect them into a single array observable. This cannot be done with a conventional instance, since each observable would have to connect to its own oracle, and the number of oracles is fixed. Because of this limitation, the instance has a dynamic set of oracles. There is always one oracle that is undefined, if a connection is made to this oracle, another oracle is spawned "underneath" it. This way, observables can be connected to the instance indefinitely.

An alternative strategy for achieving this, is an array aggregate that takes an integer parameter, and has that many oracles. This would mean

if a new observable was to be added to the array, the instance would have to be deleted, and re-instantiated. In either case, the precise detail could be implemented by the user interface, and the modeller would only need to know that he is collecting many observables into an array.

6.6 Algorithms

There are a number of interesting algorithms in the Asylum. The last chapter focused on the algorithms used to implement type checking and propagation. Another algorithm is used to propagate data through the Asylum, to resolve dependencies.

Instances are able to send and receive specific messages. A message can be sent to an instance's oracle to tell it that the data has been updated, for example. These messages are used for the type propagation algorithm, as well as the data propagation algorithm. This message sending is actually implemented with a function call, to send a message to an instance, you call a function, specific to that instance's entity, specifying the advice, oracle, and message that you want to send. The actual action carried out by the instance, depends on its internal state. Thus the messages can be thought of state transitions.

When agency occurs, and the value of an advised observable changes, the following sequence of actions occurs: Firstly a message is sent to all oracles that depend upon this advice, telling them the value is about to change. The dependent instances then send more messages to *their* dependents, until this dirty signal has propagated into every instance that will be affected by the change. After this, the value is changed, and another message is propagated through the model, instructing instances to re-evaluate their oracles. Because it is possible for the flow of information to *fork*, and then come back together again (in general, models are directed acyclic graphs), some instances may have two or more of their oracles "dirtied" by the "will change" signal. In this case, these instances must not re-evaluate their oracles and re-compute their advice until all of the affected oracles have been declared "stable" by

their dependencies.

This mechanism acts in a similar manner to a mutex, causing an instance to wait until its oracles are stable before using the contained observable data. A similar, but distinct method is used to implement the clocking agency, a dirty signal is sent, followed by a stabilising signal. There are also signals to inform instances that their observables are being linked, or unlinked, and thus the instances can propagate the undefined value through the model. Ultimately, the predefined instances have the final say in whether or not a "compromise" signal (used to inform instances of broken dependency) is sent, so it is possible to design fancy instances that use the "@" value for reasons other than a broken dependency. One example is the "`var_default`" instance, another is advising the "@" value, if we try and look up an array index that is out of the bounds of the array.

Algorithms were never written to detect infinite dependency, or race conditions. It should not be too difficult, however, to do this. It is just a matter of recursively spanning the network, looking for an instance that indirectly depends on itself, and looking for a pair of instances linked with an evaluation dependency, that both indirectly depend on the same source of agency. The main issue with these algorithms is the time they take to compute, but because they are computed at "modelling time", rather than while the model is animating, this is not as serious a problem as it is in Eden.

6.7 Compiling and using the Asylum

Compiling the Asylum package requires development libraries for GTK+ 2.0, libglade-2, Glib 2.0 and libreadline.

Unpack the source archive, enter the asylum directory and type "`make`". This should compile all the libraries and executables that are known to work well. The reader will most likely be interested in running the program "shiver", which makes use of the libraries "libhive.so" and "libasylum.so". This can be executed like so:

```
spark@stealth:~/asylum$ LD_LIBRARY_PATH=lib bin/shiver
```

This allows the shiver executable to locate the libraries it needs. Observe the *.hv scripts in the "share" directory, for some of the possible commands that can be entered into the shiver shell. Beware that the lexer is not very good, and does not strip off whitespace present at the end of command lines!

To execute the gtk blocks demo, which uses a model to internally guide the motion of two blocks, move to the share directory, and execute the following:

```
spark@stealth:~/asylum/share$ LD_LIBRARY_PATH=../lib:../lib/plugins \
                                                    ../bin/9.gtkblocks.bin
```

The documentation is generated with the "make" command, and can be found in the doxygen_output directory. The Doxyfile present in the root of the asylum package is suitable for the version of doxygen present on DCS machines at the time of writing. In the future, it may be necessary to upgrade this file using the doxygen executable installed on the system.

Chapter 7

Future Directions

7.1 Graphics

Eden has supported the two notations "Scout" and "Donald" for some time now. Together they are a powerful tool for implementing graphical environments such as user interfaces, and symbolic representations of observable structures within the model.

It is possible to implement similar functionality within the Asylum, but instead of being a notation, the system would revolve around special data types and predefined entities. A predefined entity could load a raster image from disk, further entities could process it in several different ways, involving cropping, stretching, combining with other images. Finally a predefined entity could draw the raster image onto the screen, in the form of a conventional window.

To implement the vector graphics functionality of donald, we could use an array of polygon types, which could be built up using array entities, dependent upon integer coordinates. This could then be rendered into a raster image and displayed as explained above.

In order to implement this, we could link the Asylum with a graphical toolkit such as GTK+. To get user input, we could have an advice of the window predefined instance, that supplied the current mouse coordinates, and the last clicked mouse coordinates. We could set up instances to depend

on these instances in the same way as is often used in Scout.

7.2 User Interface

At the moment the "shiver" shell suffers from many of the usability problems that the original "am" implementation was criticised for in section 2.5. This interface to the Asylum does not do justice to the Asylum back-end, however, and it should be possible to create a much more usable user interface, with plenty of feedback and model visualisation.

One such interface could be a "canvas" style of interface, where nesting entities are designed by "drag and drop"ing instances onto a nesting entity, and connecting them together with arrows. This interface could display the current value and type of every observable.

Much could be said here about the role of graphical "language" in the design of computer software, indeed a formal study in Semantology might be recommended. On the surface, it seems rational to believe that any such symbolic language should be as expressive as any syntactic language, and should be no slower to work with. In fact there are many domain specific applications of this idea, see Klogic[11], a digital circuit designer, and Labview[12].

Syntax has unbounded expression, since you can just keep executing commands, building upon the hidden state. Syntax, however can be very abstract, and precise, and hence not as approachable as graphics.

Certainly, we can say that a graphical application is much harder to develop, since this area of understanding is so immature. It may take some time to get the interface quite right. If successful, however, the benefits in terms of model visualisation would be enormous.

7.3 Optimisations

There will come a time, when models are sufficiently large that the amount of CPU time required to animate them, gets in the way of the construal.

In traditional programming paradigms, much of the program is static, and things like constant variables can be used to optimise the native code at compile time. This is obviously not possible in Empirical Modelling, and if it becomes necessary to compete with traditionally developed software (as Java has succeeded in doing), then here are suggested some techniques for optimising the algorithms within the Asylum.

There are two stages present, that which the model will do by itself, automatically (to represent some animated motion within the construal), and that which the modeller causes by making changes to the model. All of the optimisations presented here rely on the presence of an option to temporarily mark an entity "immutable". An entity treated in this way would behave just like a predefined entity. It would be impossible to modify, and impossible to look inside. This would obviously harm the tool's effectiveness as an Empirical Modelling tool, *but*, we would leave out the entities that we are focusing our attention on. The modeller cannot observe everything at once, so it makes sense to make transparent optimisations in the parts of the model where the modeller is not looking.

Optimisations in the dependency resolution stage could be made by using a JIT ("Just In Time") compiler. This would compile the instances and dependencies in a model, into a few assembly instructions, with a massive reduction in overhead.

Optimisations in the linking stage would consist of improving the performance of the type release algorithm, the circular dependency checking, and the checking for race conditions. The only way to do this (aside from coming up with a better algorithm) is to pre-process the nesting instances that are being closed. In this way, we save continually passing through the same parts of the graph.

We must be careful not prematurely optimise the system however. It is far more important to demonstrate that the system is sufficiently expressive in terms of representing state as experienced, and it will be quite some time before models get complex enough to put stress on the hardware of modern computer systems.

7.4 Advanced Dependency

There is a limitation in the concept of dependency. This is illustrated when we try and use dependency to do operations on arrays (which by nature have arbitrary length). The problem is that a dependency specifies a finite, bounded amount of computation. In order to perform computations on an unbounded data-structure, we need an unbounded amount of computation.

In procedural languages, this is provided with a while loop, or at a lower level, by a goto statement. In a functional style, we typically use recursion. Since definitions are so powerful at representing "cause and effect", how can unbounded computation be achieved in a definitive notation?

7.4.1 Recursion

If we take inspiration from functional languages, we can see that recursion provides a representation of unbounded computation. This can be implemented in the Asylum by nesting an instance of a nesting entity, in the entity itself.

In order to do this, we need a base case, and some form of pattern matching that causes the the base dependency to be resolved instead of the recursive entity. We also have to somehow guard against infinite recursion. Figure 7.1 illustrates an implementation of Euclid's algorithm with a recursive nesting entity.

The problem with recursion is that it is at best intricate and bizarre, and at worst, completely non intuitive. For these reasons it is probably best not suited to an Empirical Modelling tool.

7.4.2 Higher Order Agency

We do not have to use recursion in functional languages, in order to process lists, however. Many functional languages have a concept known as "higher order functions" that maps well onto our definitive notations. The idea is to produce a very reusable component, that factors away the unboundedness of

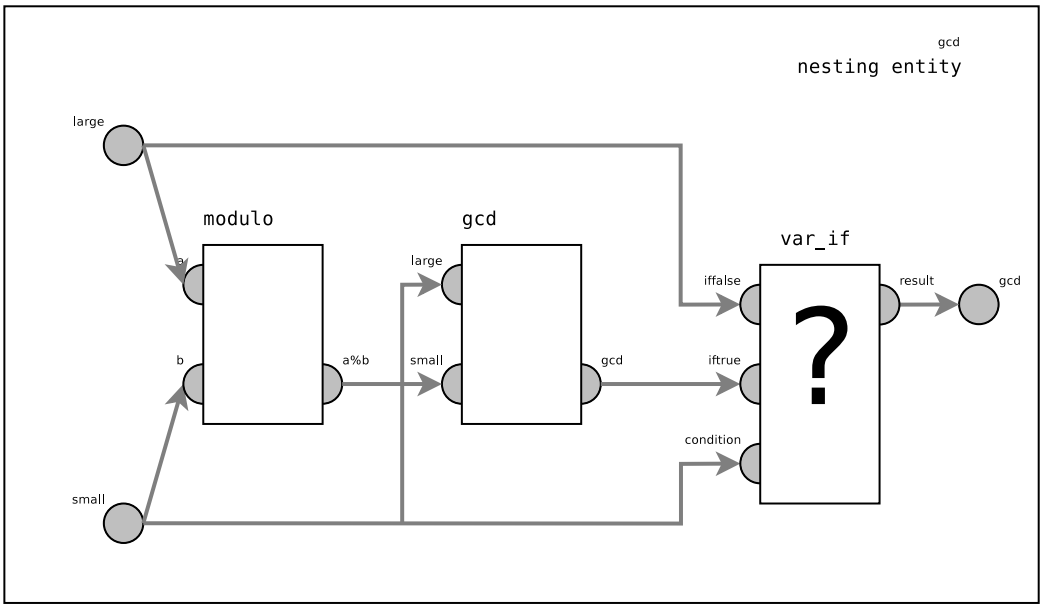


Figure 7.1: An implementation of Euclid's algorithm. Note the base case is when "`small`" is 0, i.e. `false`.

the computation as much as possible. It turns out that many array operations fall into a few categories. Here are some examples:

- Mapping is the act of applying the same behaviour to every element in an array.
- Filtering is the act of applying a predicate to every element in an array, in order to create a smaller array.
- Folding is the act of taking a list and producing a single value that is derived from every element of the list (such as an integer total).

In order to create a predefined instance that can factor out this functionality, we need to be able to supply some *behaviour* to the instance, as well as the list observable itself. This means that the instance is not depending on just the list, but also another instance, which it is applying to each of the elements of the list (in the case of map). In conclusion, to support this idea,

we need instances to be able to advise themselves, as observable behavioural specifications.

This is, in the author's opinion, the best way of solving this problem.

7.4.3 Procedural Code

The way Eden solves this problem is to use procedural while loops to iterate through lists. This could be implemented in the Asylum, in the following manner.

First, we require a special predefined instance, with two oracles, and one advice. The first oracle is of `string` type, and contains a program, written in a special language. The second oracle would contain data that the program could read (but not write to). The advice would be the output from the program.

The language in which the procedural code could be written, could be almost anything. We could use Eden's notation or something that already exists for this purpose, like Perl, or Lua, or PHP. We could even invent and interpret our own special language for this purpose.

This idea might be useful if some algorithms are simply best expressed with procedural code.

Chapter 8

Conclusion

We have studied the ADM, and its original implementation, "am", and discovered many flaws in its design. We have also endorsed and extended parts of the ADM. Of particular success has been the nesting entity idea, and type checking. The evaluation depedency idea, that allows us to implement temporal dependency, has shown to be quite expressive, but might be too abstract for Empirical Modelling.

The new strategy of partitioning the model into nesting entities has proven to be very easy to implement, and has increased the managability of Empirical Models. It would most likely be beneficial to implement this feature within the Eden tool. Static type checking has proven to be both possible, and extremely useful, but however harder to implement. Because of its design, it may be impossible to implement a static type checking function within Eden, which is a shame, since it makes the tool more usable.

The idea of using graphics instead of language, as a medium for modeller interaction, is probably too radical for inclusion in an Empirical Modelling tool at this time, but it does seem to show some promise for the representation of model behaviour, and the finer grain of modeller agency. We can conclude that this idea deserves further attention, and definitely a proof of concept implementation, maybe extending what has already been achieved with the Asylum.

Many parallels have been discovered between the technical side of defini-

tive notations, and that of functional programming languages. Especially in the field of adaptive functional programming. Ideas such as higher order agency can increase the effectiveness of definitive notations while keeping abstraction to the minimum. In particular, this allows the interactive mode of development that we see degrading during use of procedural Eden code. Definitely, some research[7] [8] into such adaptive functional domains as arrows and monads might be beneficial.

In terms of the direct materialistic benefit of this project, we can conclude that the Asylum, although not strictly complete, has provided a good proof of concept implementation for new ideas such as static type checking, nesting instances, evaluation dependency, clocking and the slightly different style of interaction with the model.

Whether these ideas are useful for representing state as experienced, is a difficult question to answer objectively. However, at least now there can be a direct comparison between the implementations, and it is clear what kinds of models are best expressed in the Asylum. The most useful benefit from this work, has been the reinvestigation of ideas from the ADM, ideas which although quite good, have unreasonably been left in the past.

Bibliography

[1] D. P. Cunningham. Re-implementing the "Abstract Definitive Machine". 3rd year project Specification, University of Warwick, October 2003.
http://www.dcs.warwick.ac.uk/ spark/3yp/specification.ps

[2] D. P. Cunningham. Re-implementing the "Abstract Definitive Machine". 3rd year project Progress Report, University of Warwick, December 2003.
http://www.dcs.warwick.ac.uk/ spark/3yp/progress_report.ps

[3] D. P. Cunningham. Re-implementing the "Abstract Definitive Machine". 3rd year project Presentation, University of Warwick, March 2004.
http://www.dcs.warwick.ac.uk/ spark/3yp/presentation/

[4] Mike Slade. Definitive Parallel Programming. MSc thesis, University of Warwick, April 1990.
http://www.dcs.warwick.ac.uk/research/modelling/hi/theses/mslade/index.html

[5] Robin Milner. A Theory of Type Polymorphism in Programming. 1978.

[6] Benjamin C. Pierce. Types and Programming Languages.

[7] Arrows, Robots and Functional Reactive Programming
http://www.haskell.org/yampa/AFPLectureNotes.pdf

[8] Functional Reactive Programming with Arrows
http://www.haskell.org/yampa/

[9] Turner, Simulation of 5 a side football using LSD and ADM, University of Warwick, 2000
http://empublic.dcs.warwick.ac.uk/projects/footballTurner2000/

[10] Joanna Pavelin, Interactive simulation of a cruise control system, Univeristy of Warwick, 2002
http://empublic.dcs.warwick.ac.uk/projects/cruisecontrolPavelin2002/

[11] Andreas Rostin. Klogic. An application for building and simulating digital circuits, written for KDE.
http://www.a-rostin.de/klogic/

[12] Labview, National Instruments
http://www.ni.com/labview/

> LabVIEW delivers a powerful graphical development environment for signal acquisition, measurement analysis, and data presentation, giving you the flexibility of a programming language without the complexity of traditional development tools.

Acknowledgements

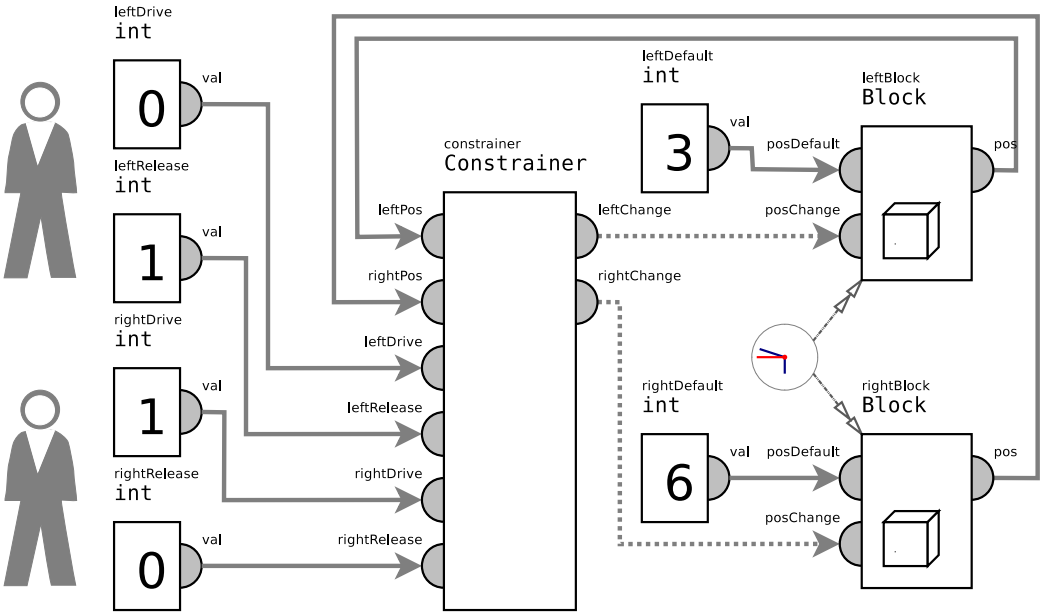
- Meurig, for answering lots of my questions.
- Ash, for showing an interest and also answering questions.
- Everyone who was involved with the EM tutorial seminars, these were very helpful!
- Karl, for sharing his experiences with Eden.
- Michael, for helping with the larger comparator network test, and supplying the incredibly simple "`ttr`" ANSI C program, which I used to help generate source code.

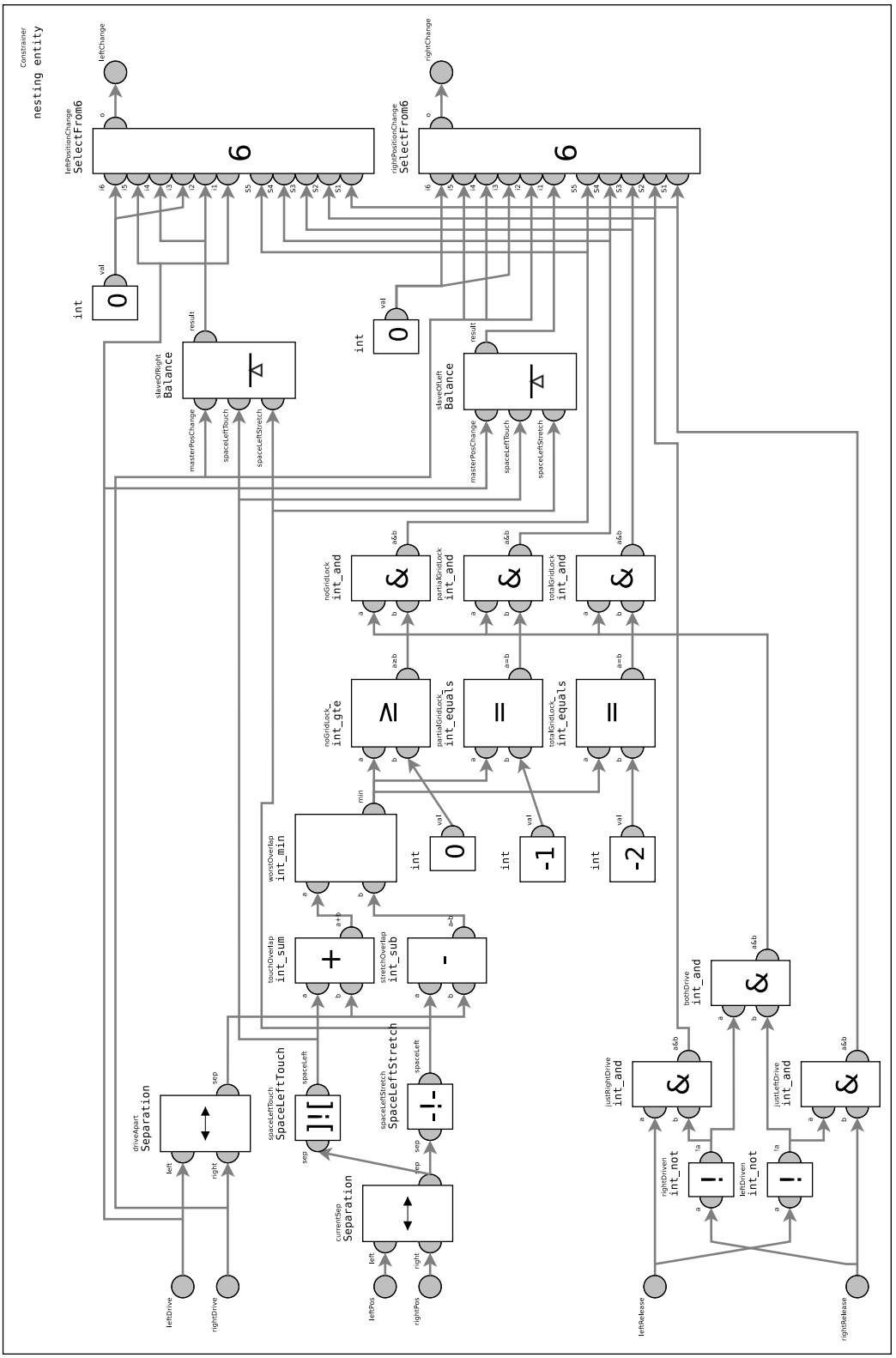
Appendix A

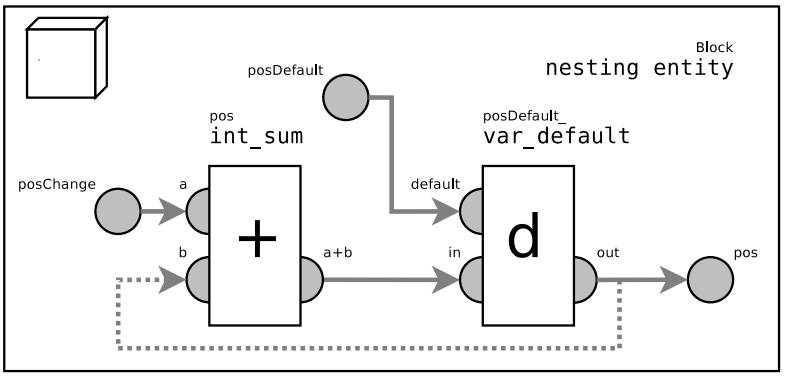
The blocks model in the Asylum

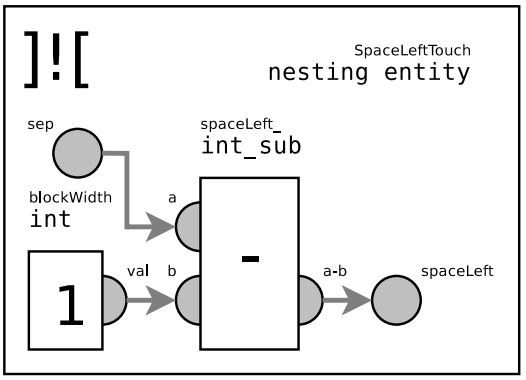
This report contains many small examples of shiver scripts, designed to illustrate certain features of the Asylum system. This, on the other hand, is a very large model (in comparison) that demonstrates the scalable nature of the Asylum. The model is presented as a series of graphical visualisations, in the same style as other parts of this document.

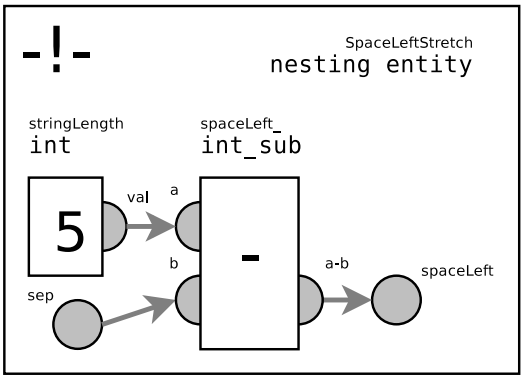
The model:

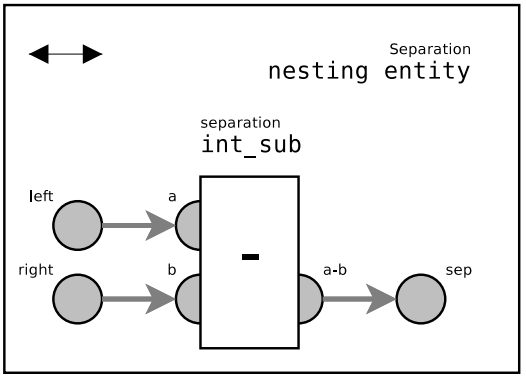


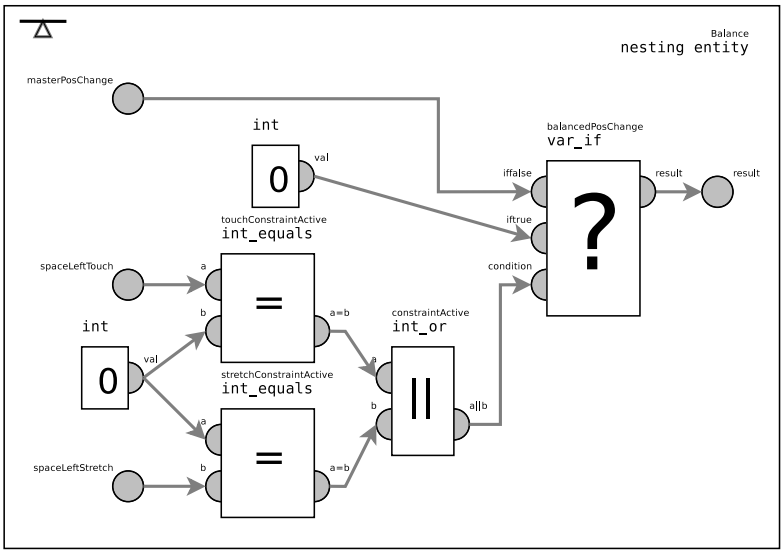

Constrainer
nesting entity
leftChange
rightChange
leftPositionChange
SelectFrom6
rightPositionChange
SelectFrom6
int
0
val
slaveOfRight
Balance
slaveOfLeft
Balance
result
masterPosChange
spaceLeftTouch
spaceLeftStretch
noGridLock
int_and
partialGridLock
int_and
totalGridLock
int_and
noGridLock_
int_gte
partialGridLock_
int_equals
totalGridLock_
int_equals
worstOverlap
int_min
min
int
-1
int
-2
touchOverlap
int_sum
stretchOverlap
int_sub
driveApart
Separation
left
right
sep
spaceLeftTouch
SpaceLeftTouch
spaceLeft
spaceLeftStretch
SpaceLeftStretch
currentSep
Separation
leftPos
rightPos
justRightDrive
int_and
justLeftDrive
int_and
bothDrive
int_and
rightDriven
int_not
leftDriven
int_not
leftDrive
rightDrive
leftRelease
rightRelease

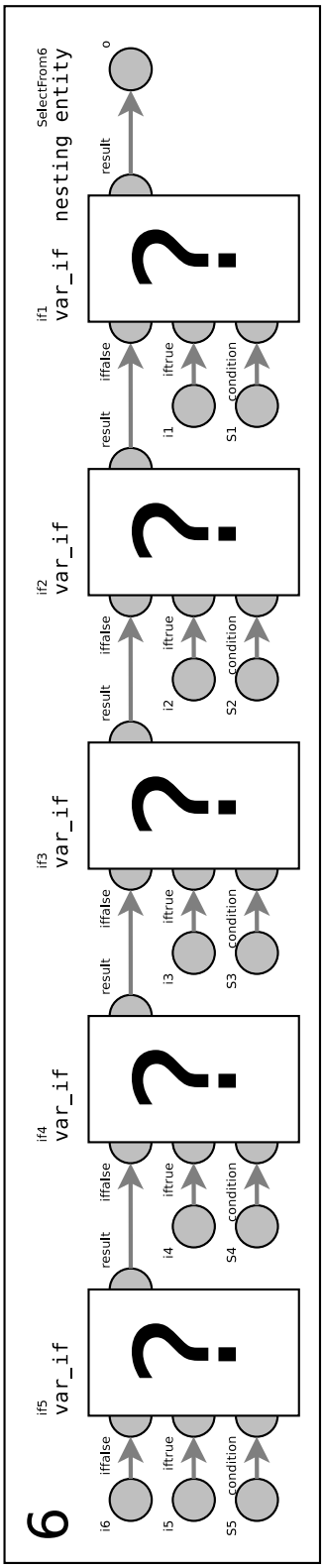

Block
nesting entity
posDefault
pos
int_sum
posDefault_
var_default
posChange
a
b
+
a+b
default
in
d
out
pos


]![
SpaceLeftTouch
nesting entity
sep
spaceLeft_
int_sub
blockWidth
int
a
1
val
b
-
a-b
spaceLeft


-!-
SpaceLeftStretch
nesting entity
stringLength
int
spaceLeft_
int_sub
5
val
a
b
-
a-b
spaceLeft
sep


Separation
nesting entity
separation
int_sub
left
a
right
b
-
a-b
sep


Balance
nesting entity
masterPosChange
int
0
val
balancedPosChange
var_if
iffalse
result
result
touchConstraintActive
int_equals
iftrue
?
spaceLeftTouch
a
=
condition
b
a=b
constraintActive
int_or
int
0
val
stretchConstraintActive
int_equals
a
||
b
a||b
spaceLeftStretch
b
=
a=b


6
SelectFrom6
nesting entity
o
result
if1
var_if
?:
iffalse
iftrue
condition
i1
S1
if2
var_if
?:
i2
S2
if3
var_if
?:
i3
S3
if4
var_if
?:
i4
S4
if5
var_if
?:
i5
S5
i6