An Implementation of the Vat Programming Abstraction

Henrik Sommerland
Abstract

An Implementation of the Vat Programming Abstraction

Henrik Sommerland

When the hardware industry hit the power-wall around the year 2005, increasing computer performance through increasing the clock-speed was no longer a practical option. The only practical way to ensure increasing performance where to start using multi core CPUs. This shifted the burden of computing performance to the programmer. But with parallel programming came many problems due to concurrent programings inherent lack of causality. Reactive programming is a programming paradigm for concurrent programming that provides a layer for abstracting away the intricacies of handling external concurrent events.

In this thesis we have been implementing and examining Vats which is a reactive storage abstraction that offers to provide safety and a flexible way of governing behavior and synchronization of data flow in an event driven program. We have made some comparisons between Vats and other known storage abstractions for reactive programming and we have also used Vats to implement the generation of scale free graphs in order to see how they can be used to solve actual problems.

This thesis contains the first ever implementation and application of Vats and as such no firm conclusions can be drawn regarding their power and usefulness, but during this implementation and evaluation, no major shortcomings or issues that could not be fixed have been encountered. The brief comparison between Vats and the other known reactive storage constructs gives an indications that Vats have a greater expressive power than the constructs compared against. The issues that where resolved during the implementation of Vats have provided a better understanding of how Vats are to work as a practical tool for programmers. Even though we draw no conclusions about Vats in this thesis we do find that they are conceptually sound and that further testing and implementation would not be a waste of time.
1 Introduction

When the hardware industry hit the power-wall around the year 2005 [17], increasing computer performance through increasing the clock-speed or decreasing the transistor size were no longer viable options. The only practical way to ensure increasing performance was to start using multi core CPUs. As computers started to rely solely on multi core support the burden of computing power was shifted from the computer hardware to the programmers. But with parallel programming came many problems. Concurrent programming is inherently difficult due to the lack of causality.

One programming paradigm for concurrent programming is reactive programming. Reactive programming provides facilities for writing responsive programs in which changes in data can automatically and asynchronously trigger events on other data. The central point of reactive-programming is that the order of the events are not known a priori and the goal of reactive-programming is to provide mechanisms for the programmer to handle this. Although reactive programming does not solve any of the problems inherent to concurrent programming such as race conditions and deadlocking they do offer the programmer a set of useful tools to manage complex interactions of events.

In this thesis we will examine the Vat [6] (described in more detail in Section 3) as a reactive storage abstraction. The Vat provides a wrapper for a field of data providing triggers on writes to the field. Through the configuration of actions within the Vat it provides callbacks (enabling reactive programming), invariant preservation, dynamic behavior and some protection against race conditions. Vats are in its nature flexible and there are several variants of Vats that vary in complexity and capability.

This thesis is only the first step towards understanding how Vats work. We have, in this thesis, implemented the Vat in the Encore [5] language. We have also tried to solve any issues and ambiguities regarding the semantics of the Vat. We will only attempt to implement the basic Single Valued Vat and the Multi-Valued Vat\(^1\) even though there are other flavors of the Vat, this is due to time constraints and that the more complicated Vat flavors are predicated on the existence and understanding of the basic Vats. We will also briefly discuss the Vat in relationship to some previously studied desirable properties for reactive-programming [1]. We have applied our implementation to the preferential attachment problem as a small preliminary examination of an application of Vats. No in-depth analysis or comparisons of the Vat have been performed in this thesis.

Since a big part of this thesis concerns ambiguities and issues regarding the semantics of the Multi-Valued Vats we make no attempt to formalize\(^2\) of the

---

\(^1\)Described in section 3.2
\(^2\)No formal semantics for the Multi-Valued Vats where presented in the original paper by Clarke and Wrigstad [6].
semantics of the Multi-Valued Vats. This is due to time constraints and the fact that there is only a simplified semantic for the Single Valued Vat presented in the Vats paper [6] and the formalization of the Multi-Valued Vats may require a complete formalization of the Single Valued Vat as well.

2 Background

In this section we will cover some background information pertaining to the rest of this work. We will give a brief introduction to the encore programming language, the active object model and reactive programming.

2.1 Encore

Encore is an imperative, object-oriented language based on the active object model that is being developed\(^3\) at Uppsala University. Encore was designed with the goal of being “concurrent by default” in the sense that concurrent execution should be the default and sequential execution would need to be explicitly requested.

There are two kinds of classes in Encore, passive and active classes. Active classes produce active objects and all method calls between two active classes gets sent as messages and will be executed asynchronously, and every method of an active objects will have its return value wrapped in a future (see Section 2.2.1). All fields in an active class are private and can not be directly accessed by other object but all methods are public. Passive classes do not produce active objects and method calls to them gets execute sequentially and the caller will execute the method, and effectively block until the method is finished. All fields in passive objects are public and can be both read and modified by any object that has a reference to it. Even though passive classes introduce the possibility of a shared state, their existence is still justified due to the overhead from doing message sends and creating futures.

Encore also has built in support for some advanced features such as Traits [16] for enabling separation of concerns without introducing inheritance, parallel combinators [5, 11] for automated distribution of parallel computations and reference capabilities [5, 10] to ensure freedom from data races.

2.2 The Active-Object Model

The active object model is a concurrency model for object-oriented programming. Each active object in the active object model has its own message queue

\(^3\)Encore is, at the time of writing, under constant development and the description provided here might not be accurate at the time of reading.
and method calls to active objects are turned into asynchronous method sends. The active objects typically process their messages sequentially. If the language supports concurrency then each object can execute its own message on a separate thread and through this the calling object does not need to wait until the callee has finished processing the message. If all fields in an active object are private and can only be accessed through method calls, data races between active objects are avoided since all fields are protected because each 4 message is processed sequentially within the active object.

The drawback of the active object model is the overhead and latency from doing message sends instead of plain method calls. The benefits of the active object model is that the parallelism happens “by default” since the execution environment is free to schedule execution in parallel.

In Example 1 shows an Encore (described in more detail in Section 2.1) program that calls two methods from the active classes A and B multiple times but always in the same order. Looking at the output we see that the execution of the methods foo and bar interleave in a nondeterministic way even though the caller always calls the methods in the same order. This is because the method calls on a and b may be executed on different threads.

```
Source
1 class Main
2   def main(): void {
3     let a = new A();
4
5     for i in [1..5]{
6       a.foo(i);
7       print 10;
8     }
9   }
10
11 class A
12   def foo(x: int): void {
13     randomDelay();
14     print x;
15   }
```

Output

```
10
10
10
10
10
10
10
10
10
1
2
3
4
5
```

Example 1: Asynchronous method calls in the active object model.

### 2.2.1 Futures

Futures [2] are an important concept in the active object model. A future is a place holder that contains a value that will at some point in the future be

---

Note that this only holds when the two active objects do not share references to any passive objects.
made available, at which point the future is said to be fulfilled. In the active object model all method calls on active objects return a future. The future is returned immediately by the callee but the future will not become fulfilled until the callee has finished executing the message. This enables the execution of a method within an object to continue after a method on another active object has been called. The execution continues until the value in the future is needed and the object that owns (the caller) the future calls \textit{get} on the future. The caller will then block until the future has been fulfilled, and at that point \textit{get} will return the value of the future.

Example 2 shows how futures can be used to delay blocking until the point where it is absolutely necessary. The program only cares about the return value of the call \texttt{a.foo()}. Now since this method involves a longer computation than \texttt{b.bar()} we can call \texttt{foo} first and store the returned future. We can then call \texttt{bar} we then call the \texttt{tar} method and pass it the future \texttt{res} and we use the future in the \texttt{tar} method by calling \textit{get} on the future.

Futures also enable chaining [12], or pipelining. We can chain a function on a future, this means that this function will be executed as soon as the future is fulfilled and take the value of that future as its argument.

Example 3 shows chaining in an 

\begin{verbatim}
Example 2 shows how futures can be used to delay blocking until the point where it is absolutely necessary. The program only cares about the return value of the call \texttt{a.foo()}. Now since this method involves a longer computation than \texttt{b.bar()} we can call \texttt{foo} first and store the returned future. We can then call \texttt{bar} we then call the \texttt{tar} method and pass it the future \texttt{res} and we use the future in the \texttt{tar} method by calling \textit{get} on the future.

Futures also enable chaining [12], or pipelining. We can chain a function on a future, this means that this function will be executed as soon as the future is fulfilled and take the value of that future as its argument.

Example 3 shows chaining in a Encore program. The main method gets finished before \texttt{bar()} and \texttt{foo()}.

\subsection{Reactive Programming}

Reactive programming is a programming model suited for event-driven programming. The behavior of an event-driven program is governed by external events over which the programmer has no control. An example of an event-driven program is a user interface. Events would in this example be mouse clicks, key presses or switching focus. These events could happen at any time during the execution of the program and the programmer has no information a priori about the order in which these events occur. It is up to the software to be able to handle the fact that any event can come at any time and in any order. For the programmer to write a program that handles the stream of events in a direct way is very difficult, the programmer would have to manually account for the possible orders in which the events could occur.

Reactive programming offers an abstraction layer to help programmers deal with concurrent event-driven programming. In reactive programming the events still govern the execution of the program but the programmer needs to pay less attention to the ordering of the events. In event-driven programming callbacks

\footnote{Although some languages implements one-way message sends where no value is returned.}

\footnote{In this case we will not handle the future returned. This becomes semantically equivalent to a one-way message send.}

\footnote{In \texttt{encore} \texttt{(a : t) -> f(a)} is the syntax for an anonymous function and \texttt{~~>} is the syntax for chaining a function on a future.}
```python
class Main:
    def main() -> void:
        a = new A();
        b = new B();
        c = new C();
        print "Starting";
        res = a.foo();
        b.bar();
        c.tar(res);
        print "Main done"

class A:
    def foo() -> int:
        longComputation();
        print "Long computation done";
        2

class B:
    def bar() -> void:
        shortComputation();
        print "Short computation done"

class C:
    def tar(x : Fut int) -> void:
        print "Waiting on the future";
        y = (get x) + 1;
        print y
```

Example 2: Blocking on futures. The future returned by a.foo() on line 7 is not used until the method tar calls get on the future.

(also called actions) are attached set up to trigger on events or changes to fields that are being monitored.

Figure 1 shows a mock-up of a drawing program written using reactive programming where we have several different entities. We have the mouse, the keyboard, a canvas, a tool selector and the display output. Here the tool selector listens to event from both the mouse and from the keyboard, The canvas listens to events from the mouse and from the tool selector, and the display output listens for events from the canvas.
Source

```python
class Main
def main() : void{
  let a = new A();
  let b = new B();
  b.bar() ~> \(x: int\) -> (print x+1);
  print "Chained on bar.";
  a.foo() ~> \(x: int\) -> (print x+1);
  print "Main done."
}

class A
def foo() : int{
  slowComputation();
  print "Slow completed";
  0
}

class B
def bar() : int {
  fastComputation();
  print "Fast completed";
  1
}
```

Output

```
Chained on bar.
Main done.
Fast completed
2
Slow completed
1
```

Example 3: Chaining on futures

3 Vats

We will now describe the Vat which is a reactive storage abstraction conceived by Clarke and Wrigstad [6]. A Vat acts like a wrapper around a variable and provides actions to be attached to that variable. These actions are then fired on every attempted write to the Vat. A Vat provides four different types of actions:

- pre actions that enforce a pre condition on all writes.
- fail actions that are used for callbacks if the write is rejected.
- transformation\(^8\) actions that ensures that the invariant of the field is maintained.
- post actions that are used for callbacks if the write where to be successful.

A Vat will reject all writes that do not satisfy the preconditions imposed by the pre actions. An attempt to read a Vat will block until the value in the vat

\(^8\)For a Vat containing no transformation actions the default behavior is identical to having an transformation that is just the identity function.
has been initialized. Figure 2 shows how the value to be written propagates through the actions.

Consider that we have an application that keeps track of a company’s personnel. In some data structure representing an employee there is a field corresponding to the salary of that employee. We want to be able to asynchronously update the salary but we also have some constraints on the updates and also some actions that need to be taken if the salary reaches certain levels. Assume that the following conditions apply to each attempted change of the employees salary.

1. The salary must always be positive.
2. Both the union and the employer must agree that the salary change is allowed.
3. The tax authorities must be notified if the salary is changed by more than 5%.
4. The salary is always rounded up to the nearest integer.

This could be solved using a Vat in the following way. Register a pre action that checks that the value is positive and not zero. Add two additional pre actions that checks if the union and the employee agrees on the salary. Add a transformation action that rounds the value to be written up to the nearest integer and finally add a post action that checks if the salary has changed by more than 5% and in that case notify the tax authority.
3.1 Vanilla Vats

In the previous example we used the so called Single Valued Vat, this is the most basic kind of Vats.

3.1.1 Types

For a Single Valued Vat containing a value of type $\tau$ we have the following types\(^9\) for reading and writing:

\[
\begin{align*}
\text{read} :: & \text{Vat } \tau \to \text{boolean} \\
\text{write} :: & \text{Vat } \tau \to \text{Future } \tau
\end{align*}
\]

In the Single Valued Vat the actions are sets of function with the following

\(^9\)The names of these types differ from the ones used in the original paper [6] for clarity.
types:

\[ \text{Present } \tau = \text{Undefined } | \text{Defined } \tau \]

\[ \text{Status } = \text{Keep } | \text{Deregister} \]

\[ \text{PreAction } \tau = (\text{Present } \tau, \tau) \rightarrow (\text{Status}, boolean) \]

\[ \text{TransformationAction; } \tau = (\text{Present } \tau, \tau) \rightarrow (\text{Status}, \tau) \]

\[ \text{PostAction } \tau = (\text{Present } \tau, \tau) \rightarrow \text{Status} \]

\[ \text{FailAction } \tau = (\text{Present } \tau, \tau) \rightarrow \text{Status} \]

**Present** is used to indicate whether the value in the Vat has yet been initialized
and **Status** specifies if the actions should remain registered or if we should
deregister\(^{10}\) the action from the Vat. The ability to deregister an action from
a Vat enables dynamic behavior of the Vat, since we are able to alter the
behaviour of the Vat at runtime.

### 3.1.2 Execution

The execution model of the vat is as follows: Upon an attempted write to the
Vat the pre actions will be executed first. If all of the pre actions return true
the value to be written will be passed on to the transformation actions. The
transformation actions take the value to be written and transforms the before
it is published. These actions are run in order from least recently added to most
recently added carrying the newly produced value forwards, behaving like a
crosoperation. After the transformation actions have completed the field in
the Vat will be updated and made available for reading. The post actions are
then triggered and the write is completed. If any of the pre actions where to
return false the write will be rejected and the fail actions will be triggered.

The post and fail actions can be executed in parallel without any issues since
neither post or fail action have any side effect with regards to the Vat. The
pre action can also be executed in parallel but they must block until all of the
pre actions are done, or one of them returns false.

### 3.2 Multi Valued Vats

Multi-Valued Vats are a basic extension to the Single Valued Vat. A Multi-
Valued Vat holds more than one field, and all of the actions depends on all
of these fields. Thus the Multi-Valued Vat provides a construct that provides
asynchronous reading and writing that maintains a relation between all of
the fields in the Multi-Valued Vat. The Multi-Valued Vats has two different

\(^{10}\)When we deregister an action it will be removed from the Vat
semantics, Or semantics and And semantics\textsuperscript{11}. For Or-Vats all actions are run on every attempted write to the Vat but for the And-Vat the actions are only triggered after every field has gotten a new attempted assignment. If we go back to the example with the salary field from the introduction to Section 3, we could use a Multi-Valued Vat to hold all of the employee salaries. If we where to use Or semantics then every employee could independently negotiate the salary but it would only be successful if it did not violate some property of all the salaries. If And semantics were to be used no employee could not get a new salary unless all the other employees have negotiated their salaries, a bit like a collective agreement.

3.2.1 And-Vats

The And-Vat provides a set of fields that are, to external observers, atomically updated. This achieved through having the And-Vat only execute its actions and attempt a write when there are new values available to every field in the And-Vat and no new values will be published unless all of the fields has been assigned new values. Due to this the And-Vat must be and keep record of whether or not there are new values provided to the fields and only execute the actions in the case where there are new values present for every field in the And-Vat.

The nature of how this state is to be kept is not mentioned in the original paper [6] by Clarke and Wrigstad. An undefined situation regarding the semantics occurs in the case where one where to attempt to make a new assignment to a field in an And-Vat where this field already has a new value available but that value has not yet been written into the And-Vat. This issue will be discussed in greater depth in Section 5.2.

3.2.1.1 Tracking Pending Writes In order to keep state of whether or not there are any new values present to the fields we introduce a new set of fields, called the pending fields. Each of these correspond to a field in the And-Vat and holds the new value to be written to the And-Vat once all pending fields have been assigned a value. The actions fire once all the pending fields have been filled and if the pre actions succeed the pending fields will be fed to the transformation actions and then be stored in the exposed fields of the And-Vat. After every attempted write, regardless of whether or not the pre actions succeed, the pending fields will be reset to an undefined state.

3.2.1.2 Types In order to properly define the behaviour of the And-Vat we will need to extend the types of the Single Valued Vat to account for the

\textsuperscript{11}Multi-Valued Vats adhering to these semantics will be referred to as And-Vats and Or-Vats
more complicated semantics of the And-Vat. We will here assume that we have an And-Vat with an arbitrary number of fields but restricted to one type $\tau$. There is nothing that says that all the fields in an And-Vat needs to have the same type but it will simplify the presentation of the types. Here we have $\tau$ representing a tuple in which each element has type $\tau$ even though it is not explicitly stated it is obvious that all tuples involved have the same number of elements. The following are the general types for the And-Vat:

$$\text{AndVat } \tau = (\text{Present } \tau, \text{ Present } \tau)$$

$$\text{Status} = \text{Rejected} \mid \text{Pending} \mid \text{Succeeded} \mid \text{Failed}$$

$$\text{write} = \text{AndVat } \tau \rightarrow \text{ Present } \tau \rightarrow \text{ Future Status}$$

$$\text{read} = \text{AndVat } \tau \rightarrow \text{ Future } \tau$$

The $\text{write}$ function takes a tuple $\text{Present } \tau$ in order to enable writing only to some fields in the And-Vat. The $\text{Status}$ type is used to indicate whether or not the write was rejected due to the programmer attempting to write to a field that already has a pending value. The write is pending because not all of the pending fields have been assigned or whether or not the write was successful if all pending fields where assigned and a write was attempted. These are the types for the actions of And-Vats.

$$\text{PreAction} = \text{Present } \tau \rightarrow \tau \rightarrow (\text{Status, boolean})$$

$$\text{FailAction} = \text{Present } \tau \rightarrow \tau \rightarrow \text{Status}$$

$$\text{PostAction} = \text{Present } \tau \rightarrow \tau \rightarrow \text{Status}$$

$$\text{TransformationAction} = \text{Present } \tau \rightarrow \tau \rightarrow (\text{Status, } \tau)$$

### 3.2.2 Or-Vats

Or-Vats behave differently to the And-Vat in that the actions are always run on every single write to any field regardless of whether or not all fields are assigned. The Or-Vat does provide consistency over several values but it does not provide atomicity as the And-Vat does. The Or-Vat may be viewed as a less restrictive variant of the And-Vat. The semantics and the execution model of the Or-Vat is much alike that of the Single Valued Vat. The only difference between the Or-Vat and a Single Valued Vat contain a set of values is the fact that the Or-Vat provides a nicer interface for updating only some of the values stored in the Or-Vat.

### 3.2.2.1 Types

The types of the Or-Vat are more verbose than those of the And-Vat since every action needs to handle undefined values at every stage

---

12The $\text{Present}$ datatype is described in section 3.1.1
since after even a valid write some values in the Or-Vat might still be undefined. The following are the general types for the Or-Vats:

\[
\text{OrVat } \tau = (\text{Present } \tau, \text{Present } \tau)
\]

\[
\text{write :: OrVat } \tau \rightarrow \text{Present } \tau \rightarrow \text{boolean}
\]

\[
\text{read :: OrVat } \tau \rightarrow \text{Future Present } \tau
\]

These are the types for the actions for Or-Vats.

\[
\text{PreAction} = \text{Present } \tau \rightarrow \text{Present } \tau \rightarrow (\text{Status}, \text{boolean})
\]

\[
\text{FailAction} = \text{Present } \tau \rightarrow \text{Present } \tau \rightarrow \text{Status}
\]

\[
\text{PostAction} = \text{Present } \tau \rightarrow \text{Present } \tau \rightarrow \text{Status}
\]

\[
\text{TransformationAction} = \text{Present } \tau \rightarrow \text{Present } \tau \rightarrow (\text{Status}, \text{Present } \tau)
\]

3.3 Other Vat Flavours

In the original paper by Clarke and Wrigstad [6] three other kinds of Chocolate Vats are briefly introduced. In this part we will give a short description of these additional Chocolate Vats. These are not described in depth in the paper. So much of how they work is still not known.

3.3.1 Linked Vat

Linked Vats provides so called links between Vats that enables synchronization between Vats that are not co-located. For a group of Linked Vats a write to any of the Vats in the group will not be published until there are values ready to be published in every Vat in the group. Linked Vats can be likened to an And-Vat where all of the fields are not co-located. Although the Linked Vat provides no consistency checking for the values in the other Vats since they are no actions for the Linked Vat.

3.3.2 Nested Vat

Nested Vats are Multi-Valued Vats in which each field is a Vat. The Vats within the Nested Vat are connected with links as in the Linked Vat and as such they can be considered as Linked Vats with the extended functionality of providing consistency checks for the values that are written into the Vats. The Nested Vat comes with its own set of actions and has And and Or semantics analogous to those in the Multi-Valued Vat for determining when the actions are to be executed. For both semantics the inner Vats pre and transformation actions are run first and then the vats of the Nested Vats actions are run. For the writes to be published both the inner and the outer Vats pre actions must succeed.
3.3.3 Type-changing Vats

Type changing vats are just regular Single Valued Vats or Multi-Valued Vats where the type of the value written, the value stored and the value returned can differ. The semantics of the type-changing Vats does not differ from the regular Single Valued Vat or Multi-Valued Vat in any way.

3.4 Vats as stream filters

One possible way of interpreting the behavior of the Vat is as a stream filter. In a normal reactive program, one entity can listen to a field and be notified on every update. This essentially turns the field into a kind of stream, the listening entity will get a stream of the values written to the field in the same order as they are written.

The Vat extends this notion by creating a form of filtered stream. Now the stream generated by the writes to the field can be filtered in such a way that the values produced will adhere to certain invariants and the Vat provides an additional stream for listening to failed writes. Figure 3 shows a simple schematic representation of the vat as a stream filter. Not only does a vat provide two streams for both failed and successfully "filtered" values. Since the Vat still provides explicit readable it makes the outputted stream also pollable, so consumers who only need the value of the Vat at a specific time can simply just read the value and does not need to setup a callback.

![Diagram of Vat as a stream filter]

Figure 3: Here we can see schematic representation of the Single Valued Vat as a stream filter. a) Is the stream of failed writes to the Vat. b) Is the readable value of the vat, which is not a stream. c) Is the stream of successfully filtered and transformed values. d) Is the stream of writes to the Vat.

This interpretation extends to the Multi-Valued Vats in a intuitive manner. The Multi-Valued Vat can be interpreted as a stream multiplexer. Writes to several fields are converted into a single stream in which an invariant has been enforced on all values. The difference between the And and Or Multi-Valued Vat in this
interpretation is when new values are pushed onto the output streams. For the Or-Vat a new value will be pushed to the stream on any change (that satisfies the pre conditions) to any of the values but for the And-Vat a new value will only be made available when all fields have been updated.

Figure 4: A mockup of a drawing program where the different components are interconnected with Multi-Valued Vats.

Figure 4 shows the example of a drawing program from Section 2.3 but redesigned using Vats. Here we are using the following three Vats

1. Is a Or-Vat that the tool selector uses to subscribe to mouse clicks that are in the region of the tool selector and keyboard events that correspond to hot-keys.

2. Is an Or-Vat that is used by the canvas to filter out mouse clicks that are not within the canvas and that each mouse click is happening when there is a tool attached to it. This Vat can also transform the coordinates from the mouse to scaled coordinates for the canvas to use directly.

Even though the diagram is more complicated using Vats than the diagram in Figure 1 but it will offload some of the work from the canvas and the tool selector since both the canvas and the tool selector will only receive events that are relevant to them and are events to which they have a defined response.

4 Related Work

In this section we will go through some of the previous work that has been done examining the various properties of reactive programming. We will also
take a short look at some other reactive constructs that bears similarities to the Vat.

4.1 The Taxonomy Of Reactive-Programming

In their paper “A Survey On Reactive Programming” [1] Bainoumoguisha et al define a set of six properties that constitutes the taxonomy of reactive programming languages. We will now give a brief explanation of each of these properties.

4.1.1 Basic Abstractions

The basic abstractions of a reactive programming language is the most primitive form of reactive mechanism provided for writing reactive programs. There are behaviors that correspond to continuous time-varying values. The other basic form of reactive abstraction is the event which constitutes a discrete value being presented at a specific point in time.

4.1.2 Evaluation Model

The evaluation model of a reactive program corresponds to how the program reacts to events or changes of time-varying values. There are two different evaluation models for this: push-based and pull-based evaluation. Push based evaluation is where the entity in which the event originates “pushes” the change to the entity that needs this information. Whereas for the pull based evaluation model the entity who needs the information must manually request or read the information.

4.1.3 Glitch Avoidance

A glitch in the context of reactive programming is an inconsistency between data caused by changes not being propagated through the program in the correct order. Bainoumoguisha et al. provides the following example in their article [1]:

```plaintext
1  var1 = 1
2  var2 = var1 * 1
3  var3 = var1 + var2
```

In this example we assume that var1, var2 and var3 are all reactive variables and changes to any of them will be propagated throughout the entire program. In the example we can see that var3 should always be equal to 2 * var1. But imagine a situation in which var1 is changed to 2 and this change causes var2 and var3 to be recomputed. There exists a possibility that var3 will be evaluated
before var2 and in that case var3 will get the value 3 since it still sees var2 as being 1. This is what is referred to as a glitch.

4.1.4 Lifting

Lifting refers to the languages ability to perform operations on reactive entities such as time varying variables or event sources. In the example in the section above we can see implicit lifting in use. In that example the arithmetic operators can operate directly on the reactive values without any additional effort. Another method to achieve lifting is through explicit lifting, in which there exists some operator that lifts some operator or function to work on behaviors or events. There is also the notion of manual lifting for which the programmer must manually extract the current value from a behavior or block until a new event occurs. Languages that support overloading of operators and functions to work on behaviors and events fall in to some kind of gray area between implicit and explicit lifting.

4.1.5 Multidirectionality

Multidirectionality is the ability to have events or time-varying variables trigger changes of other reactive values. An example presented in the paper by Bainoumogunisha et al [1] is that of having two time varying variables that holds the current temperature. One is in degrees Celsius and the other in degrees Fahrenheit. Multidirectionality would enable the behavior that when the Fahrenheit value gets updated it automatically updates the Celsius value with the corresponding value and vice versa.

4.1.6 Support for Distribution

This property concerns a languages ability to provide stable reactive programming facilities for distributed systems. Distributed settings where events and behaviors are located across different network nodes introduces the problem of ensuring consistency between two communicating values located on different nodes. The difficulty arises from the possibility of node failure and long networks delays.

4.2 Vat-Like Constructs

Some other languages implement constructs conceptually similar to the Vat. Some of these constructs will be described here and they all share the common property of providing mechanisms to extend variables or fields with reactive capabilities. We will briefly give examples of a few of these constructs below.
4.2.1 Cells

Cells [18] is an extension to the Common Lisp Object System. A cell in this context is an extension to a Common Lisp Object that provides this object with slots that allows for the registration of functions to be triggered on changes to these slots. To these slots observer functions can be added by other objects that fire on every write to the value in the slot changes and thus provides a mechanism for other objects to listen to changes of the cell. These slots or cells can be configured to be either writable from the outside or such that they can only be changed as a consequence of changes to other cells.

4.2.2 Trellis

Trellis [8] is a reactive-programming extension to Python in which rules are attached to special values stored in cells. Cells are tied together with rules and these rules are actions that are triggered on value changes to these cells. A rollback will automatically occur in the case of errors or inconsistencies.

4.2.3 Lamport Cells

Lamport Cells is a library that provides mechanisms for reactive programming for the E [13] programming language. Lamport cells are based around the concept of reactors and reporters. Reactors listen and react to updates from the reporters. Reporters on the other hand accepts registrations of reactors and monitors values and pushes the updates to the reporters. Reactors can be conditionally deregistered from a reporter and the reporter will then stop sending updates to that reactor.

4.2.4 Radul/Sussman Propagators

The Radul/Sussman propagator is a generic propagation model that is built on MIT/GNU Scheme [15]. This construct is based around propagators and cells. Propagators are functions that gets evaluated on changes to cells and propagators and cells are interconnected together. Cells can hold more that one field of data and the propagators support propagation and manipulation on only partial data from the cell.

5 Ambiguities And Issues

In this section we will discuss some of the issues and ambiguities that arose from the implementation and closer examination of how Vats work.
5.1 Atomic Conditional Registration of Actions

An issue that was discovered is that there exists situations in which there must be a possibility to atomically register actions given that some criteria holds. This manifests when one have to register an action after checking the value stored in the Vat.

Let us illustrate this with an example. Imagine that we have a Vat in the control system for a nuclear power plant. After some event has occurred we need to add a post action that triggers an emergency shutdown if the core temperature passes a certain threshold. If we just register the new post action as usual after the event has occurred there is a chance that in the time between the event that causes us to attempt to register the new post action, some other process writes a new core temperature to the Vat that is above the threshold. If no new successful write to the Vat occurs the new post action will never be triggered and a meltdown may occur.

We need a method to atomically read the Vat and check if some condition holds and then we register the new action and if the condition fails we execute some other action. In the example with the nuclear power plant this would give us the possibility to execute the emergency shutdown if the core temperature has already passed the threshold and otherwise just register the new post action and have it trigger the shutdown if the. With this new operation there is no risk of a race condition occurring.

Here are the type for conditional registration for post actions for the Single Valued Vat.

\[
\text{ConditionalRegister} :: \text{Action } \tau \rightarrow (\text{Present } \tau \rightarrow \text{boolean}) \\
\rightarrow (\text{Present } \tau \rightarrow \text{unit}) \rightarrow \text{boolean}
\]

The first argument is the action\(^{13}\) to be registered if the second argument, which is the condition, holds. The last argument is what is to be run if the condition does not hold. The returned boolean indicates whiter or not the condition held. Another way of doing this could be to have the action to be registered trigger if the condition does not hold, but that would be less flexible than this approach.

5.2 Conflicting Writes to And-Vats

The exact way in which writing to an And-Vat works is not clearly defined in the original Vat paper [6]. One of the more problematic issues to resolve has been how to handle conflicting writes, where an attempt is made to write a new value into a pending field that already has a pending value. There are at least four possible ways to handle this case:

\(^{13}\text{Where Action } \tau \text{ is one of the action types presented in Section 3.1.1.}\)
1. Reject the write.
2. Replace the pending value with the new value.
3. Store the new value in a queue and attempt to write it when the there is no pending value in the corresponding pending field.
4. Add a new set of actions that that handles the conflicting writes and let the programmer decide what to do.

Rejecting the write is an easy option and the entity that attempts the write could be informed about this rejection through the return value from the write method. Although this might not be desirable for a situation in which only the latest value is of any interest. The simplest alternative to rejecting is to overwrite the previous pending value. This may not be desirable in several cases since it will essentially destroy data. If overwriting were to be implemented as the default there should be some way for the programmer to opt-out of this feature. The third alternative is to buffer all attempted writes to an already occupied pending field. One would then after the other fields have been assigned and the pending field has been reset pop the first value on the queue in to its respective pending field. This solution would solve the problems of the two previous solutions but it would lead to a more complicated implementation of the And-Vat.

If one allows writing multiple values at the same time things becomes less trivial. The problem here is which of the values in the attempted write should be written into the pending fields as can be seen in Figure 5. Should the value that has no conflicting counterpart be added whilst the other is rejected, as in b? Should one reject the entire write as in a or should one overwrite the conflicting values as in c. This issue could also be solved using queues for each field and just do as we discussed in the previous paragraph.

![figure 5](image-url)

Figure 5: Example with conflicting writes to an And-Vat.
5.2.1 Solution

The solution we thought was the most flexible and powerful one was to add a new set of actions called *merge* actions to handle conflicting writes. These actions would take as its input the pending fields, and the fields that are to be written. This action could then decide how to combine these values or whether or not this write should be rejected entirely. This would give the programmer a lot of control and one could easily imagine a situation where there was no issue overriding one of the fields. Consider an And-Vat where one of the fields contain mouse coordinates, it may under some situations make absolute sense to always keep the mouse coordinate field hold the latest mouse coordinate even if the And-Vat is still waiting for its other fields to be filled in.

Adding *merge* actions to the And-Vat would also enable easy implementation of basic rejecting or overwriting as a default behavior. The only thing that would be needed to be done in those cases would be to add a predefined *merge* action at the time when the And-Vat is created that either overwrites every pending field or rejects the write in the case of a conflict.

Another question that arises is whether or not the *merge* actions should operate on the pending fields if all of them have been assigned? If one where to allow this the *merge* actions would start to resemble something more like a pre pre action transformation action. Only running the *merge* actions in the case of partial writes may be the cleaner solution since it makes it more obvious what the purpose of the *merge* actions is.

In the case of a conflicting write the *merge* actions would be triggered and if all fields are filled after the merge actions are done the new values are written as usual. If not all of the fields were to be defined the pending fields would be updated with the merged values.

The merge actions would have the type:

\[
\text{MergedQ Present } \tau = \text{Rejected } | \text{ Merged Present } \tau
\]

\[
\text{MergeAction} = \text{Present } \tau \rightarrow \text{Present } \tau \rightarrow (\text{MergedQ Present } \tau, \text{ Status})
\]

Here the argument takes as its arguments the partially assigned fields of the And-Vat and the set of pending fields. It then preforms the merge and if the merge was successful it will return the new set of pending values and if the merging was deemed to be not allowed it would return *Rejected*. Figure 6 shows a schematic overview of how the flowchart of the And-Vat would look with *merge* actions attached.

Below is an example in Encore of a *merge* action that in the case of a conflict writes the mean of the two conflicting values to the And-Vat:

```ocaml
1 \(\text{let merged_fields} = \text{new} [\text{Maybe int}](|\text{pending_fields}|)\);
```
Figure 6: A schematic overview of the And-Vat with merge actions.

See Section 6.2.1 for a description of the implementation of the And-Vat in Encore.
5.3 Exposing Undefined States In Or-Vats

Another minor ambiguity is whether or not the Or-Vat should expose any undefined states. One approach would be to have reads on the Or-Vat block until all fields have been designed but this still leaves the issue that the undefined states can be observed through the actions. If one were to buffer the writes until all fields have been assigned and then publish them after all fields has been assigned choosing the semantics becomes troublesome, should one then execute the pre and transformation actions on every write but not trigger the post until all fields have been assigned, should one then run the fail actions if the pre actions fail? This could still leak undefined states in any action since nothing prevents the attachment of an action that allows some object to observe the state. This could be remedied by letting the first stage of the write to the Or-Vat behave like that of the And-Vat, but this introduces the problem of handling conflicting writes.

If one chooses to disallow observing undefined states another issue still remains; what if the user through a read or a transformation actions resets one of the fields to an undefined state. Should this be allowed? And in such a situation what should the behaviour be. If this is not allowed transformation actions that resets a value to undefined must either cause the write to be rejected or throw a run time error.

A compromise has been chosen as the solution. If a read is attempted in a situation where some of the fields are undefined the entire Or-Vat will be considered as undefined and the read will in that case block. If the transformation actions where to reset some field that was previously defined to an undefined state an extra check after the transformation actions will make sure that this is not the case and if some value where to be reset to undefined the write will be rejected and neither the post or the fail actions. This still leaves the issue of the undefined state being observable through the pre and transformation actions but no solution to this issue has been found.

6 Implementation

In this section we will cover the implementation of Vats in Encore and briefly discuss any shortcomings of the implementation. The semantics of the Vat that has been implemented here differs in some cases from the description in Section 3 since we have applied the solutions and fixes from Section 5.

The Vat has been implemented in Encore using active objects. Due to Encore not providing support\textsuperscript{14} for algebraic data types or atoms most of the types described in Sections 3 have had to be encoded differently in Encore.

\textsuperscript{14}At the time of writing.
In the implementation of both the Single Valued Vat and the Multi-Valued Vat the sets of actions were implemented using traditional linked lists. Execution of the actions is then performed by traversing the list and dropping the nodes for which the return status is **Deregister**. This could have been implemented using generic **map**, **filter** and **fold** operations, but due to a lack of support for parametric functions, no substantial gain in readability or maintainability would have been gained by implementing **map** and **filter** exclusively for use in the Vat implementation. Implementing the execution of the more complicated actions such as the **pre** and **transformation** actions would involve a more complicated mix of **map**, **filter** and **fold** which may not lead to better code.

In the current implementation none of the actions are executed in parallel. This is partially due to tasks not being implemented in the language at the time of writing this. Whether or not the actions run in parallel does not affect the usability of Vats, it would only contribute with a possible performance gain. Since this thesis is primarily concerned with the usability of Vats parallelizing the actions is not deemed important enough to warrant the implementation of a workaround to enable parallel execution of the actions.

The atomicity of the **ConditionalRegister** function is automatically achieved through the fact that Vats are implemented as an active object and thus no new write to a Vat can happen before the **ConditionalRegister** has finished.

### 6.1 Single Valued Vat

The implementation of the Single Valued Vat was very straightforward and there were no shortcoming in Encore that complicated the implementation. We have used the Maybe type from Encore to encode the behavior of the **Present** type and the **Status** type has been encoded using a bool. The types for the actions of the Single Valued Vat had to be encoded as follows in Encore:

1. \( \text{typedef Pre_action<}t\text{> = (Maybe }t\text{, }t\text{) -> (bool, bool)} \)
2. \( \text{typedef Transformation_action<}t\text{> = (Maybe }t\text{, }t\text{) -> (bool, }t\text{)} \)
3. \( \text{typedef Post_Fail_action<}t\text{> = (Maybe }t\text{, }t\text{) -> bool} \)

Here the types for the post and fail actions have been merged into one type for simplicity. The execution of the post and fail actions have also been merged into one executePostOrFail method since their behavior is identical.

### 6.2 Multi-Valued Vat

One issue that arose during the implementation of the Multi-Valued Vat is that there is no mechanism in Encore that enables having sets of objects of arbitrary length and contain multiple types. One possible option is to use either tuples and enable support for multiple types but locking down the number of values that
could be stored in the Multi-Valued Vat. Using tuples would also complicate
the types since the types for each operation would need to contain the entire
tuple and the Multi-Valued Vat class would have needed to be parameterized
over all the possible types. The other option is to encode the set of values
in the Multi-Valued Vat using an array, this enables varying the size of the
Multi-Valued Vat and makes the types simpler, but this limits the Multi-Valued
Vat to store values of only one type.

In the end the decision to use arrays was made since using tuples would have
led to unreadable types if the size of the Multi-Valued Vat was set at a to large
number. And to provide Multi-Valued Vats of different size we would need to
have implemented them separately. Using arrays provides a good trade-off by
allowing arbitrary size of the Multi-Valued Vat whilst still keeping the types
and implementation simple. It it still possible to emulate the behavior of a
And-Vat with several types through storing a tuple in the And-Vat and then
using the merge actions to emulate the behavior of a Multi-Valued Vat with
different types.

6.2.1 And-Vat

In order to handle the fact that values only gets written to the And-Vat when
all fields has been assigned we need to introduce a second set of fields referred
to as the pending fields. Each value in the set of pending fields can either be
assigned a value or be undefined. This is encoded using the Maybe type. An
attempted write to some of these fields will only trigger the actions if every
pending field is assigned a value. Regardless of whether or not the write is
successful all the pending fields will be reset to undefined after the actions has
been run. If an attempted write where to try to assign a value to one of the
pending fields that has already been assigned a value the merge actions will
trigger and attempt to merge the conflicting values.

The type of the field in the And-Vat has been encoded as Maybe [t] and the
pending fields as [Maybe t] to encode the Present type. The types of the write
method is:

```
1 write([Maybe t]) -> (bool, bool, bool)
```

Here the tuple (bool, bool, bool) encodes the Status type. The first value indicates if
the write was rejected, the second if the write is pending and the third
whether or not the write was successful if all pending fields where assigned and
the actions where executed. There is also a singleWrite method that takes a
value of type t to be written and an index and only writes a single value to the
And-Vat. This is present simply for convenience reasons.

Apart from the merge action the types for the other actions are very similar to
those of the Single Valued Vat.

```
1 typedef Pre_action<t> = (Maybe [t], [t]) -> (bool, bool)
```

27
The return type for the `merge` actions is not descriptive and a detailed explanation is required. If the last element in the returned tuple is `Nothing` then this means that the write was rejected. If any of the `merge` actions reject the merge then the entire write is rejected and the pending fields does not change their values.

In order to enable default behavior for the And-Vat the constructor for the And-Vat has the following type signature:

```python
def init(size: int, override_behavior : bool, 
        merge_action: Maybe Merge_action<t>)
```

The `override_behavior` flag indicates whether or not overriding should be the default behavior of the And-Vat, if it is set to `false` all conflicting writes will be rejected and if it is set to `true` all conflicting fields will be overwritten by the new value. The `merge_action` allows the programmer to add a custom default behavior. If it is not set to `Nothing`, the action provided will be the first `merge` action to be added and the `override_behavior` flag will have no effect. The default actions will be permanent and will never be deregistered.

### 6.2.2 Or-Vat

The implementation of the Or-Vat is very similar to the And-Vat but without the `merge` actions and the pending fields. For the Or-Vat we do not need to pay any special attention to conflicting writes where the write would overwrite an already defined field. Since we always trigger the actions on every write the `transformation` actions can be used to handle the merging. The default case for the Or-Vat is that writes to an already defined field always overwrites the old value. In the case that a field gets reset to undefined at any point during the write, the write will be discarded and the write method would return `false`. The check to see if an already assigned field gets set to undefined are run after the `transformation` actions. This is to enable the programmer to write special `transformation` actions to resolve un-defining writes if that where to be necessary for the specific problem.

For the Or-Vat the actions has the following types:

```python
typedef Pre_action<t> = ([Maybe t], [Maybe t]) -> (bool, bool)  
typedef Transformation_action<t> = ([Maybe t], [Maybe t]) -> (bool, [Maybe t])  
typedef Post_Fail_action<t> = ([Maybe t], [Maybe t]) -> bool
```
6.3 Implementation Shortcomings

One issue with the implementation is that it is not possible to achieve the behavior that reads block until the Vat has been initialized. An active object can only process new messages after the current method has finished. So if the read method would not return until the Vat had been initialized, it would deadlock since any message that writes to the Vat would not get processed, since the read method never terminates. The suspend keyword that Encore provides circumvents this issue by putting the current method back onto the message queue and processes the messages that are currently on the queue before continuing to execute the method that suspended its execution. Although suspend enables an active-object to continue to consume messages before returning from a method this is not suitable to this problem since it does not guarantee that the value returned by a read would be the first value written to the Vat. In order to enable this blocking behavior for reads the implementation would need to be able to keep explicit track of what state the object is in and handle the messages accordingly. This could be achieved in a language where one has full access to the message queue like Erlang but no such functionality exists in Encore. This has been partially solved by introducing a set of readOrRegister method. These takes as their argument an action and if the Vat has been assigned it returns the value of the Vat otherwise the action gets registered. These can be considered as a special case of the conditionalRegister method.

Another shortcoming is that there might be a need to provide a mechanism to support attachment of actions upon creation of the Vat. Without this there is a risk that a write that should not be allowed may get written to the Vat because the necessary pre action has not yet been registered. Providing this functionality would mean that we would have to extend the signature of the constructor of the Vat significantly and that would obfuscate the code. It is still possible to achieve this behavior by just making sure that the object that creates the Vat does not share a reference to the vat before all the necessary actions have been registered.

As of now the Multi-Valued Vat is entirely unsafe in the sense that the values stored in the Multi-Valued Vat can be changed by any action since the arrays are passed by reference. Below is an example of a malicious pre action for an And-Vat:

```plaintext
\(\text{old\_value : Maybe [int], new\_value : [Maybe int]} \rightarrow\)
\{
    match old\_value with
    Nothing => (true,true)
    Just arr => {
        repeat i <- |arr|{
            arr[i] = 0
        };
        (true,false)
    }
}
```
This pre action will overwrite all the fields with 0 and not notify any listener through the post actions since the pre action returns false. This can be solved with reference capabilities [10], since one could then restrict the operations permitted on the arguments 15.

7 Experiment

In this section we will examine how Vats can be used to solve the preferential attachment problem.

7.1 The Preferential Attachment Problem

Preferential attachment in the context of graphs is a process that creates undirected graphs with a scale-free topology 16 in which the degree distribution for the nodes in the graph follows a power distribution.[3]

Preferential attachment works by letting each new node added to the graph pick a connection to a node at random with a probability proportional to the relative degree 17 of that node. Some real world systems that have been shown to produce scale-free graphs are among other scientific collaborations and protein networks.[14, 9]

7.2 Algorithmically Generating Scale-Free Networks

The algorithm we will use to generate a scale-free graph using the preferential attachment mechanism is based around storing the nodes and their edges in a sorted set. We keep track of the order in which we add nodes by labeling them with the order in which they were added. When we add a new node we randomly pick a node with a lower index than the new node (an older node) and then at random either connect the new node to the picked node or to one of its connected nodes. With this algorithm the number of times a node appears in the set (both as a node and as a connection) will be equal to that nodes degree. Since we always pick a connection to an older node that older node will appear as the edge of the newly added node and through this the oldest nodes will be the most likely to occur the most, whilst newly added nodes will occur less frequently in the set and thus have a lower likelihood of being picked as a new connection. Through this the degree distribution will approximate a power distribution and thus the graph topology will be an approximation of a

15At the time of writing work is underway to introduce reference capabilities in Encore.
16Scale-free graphs are here assumed to be scale free but there are models for directed scale-free graphs [4]
17How many other nodes that node is connected to relative to the total number of connections in the graph.
scale-free topology. This algorithm can be extended to allow for multiple new connections per node but in that case attention needs to be paid to ensure that no double connections occur.

Below follows pseudo code\textsuperscript{18} for the algorithm when each new node gets only one connection:

\begin{verbatim}
Data: let \( n \) be the number of nodes to be added.
Data: Initialize an array \( A \) of edges with one element 0.
\textbf{for} \( i \text{ from } 2 \text{ to } n \text{ do} \)
  Select random \( j \) such that \( j < i \);
  if \( \text{Pick edge} \) then
    append \( A[j] \) to \( A \);
  else
    append \( j \) to \( A \);
\textbf{end}
\textbf{end}
\end{verbatim}

For the case where each gets more than one connection the algorithm becomes as follows:

\begin{verbatim}
Data: Let \( m \) be the number of connections each new node gets.
Data: let \( n \) be the number of nodes to be added.
Data: Initialize a 2D array \( A \) of edges with one element being the array [0 .. \( m \)].
\textbf{for} \( i \text{ from } m + 2 \text{ to } n \text{ do} \)
  Create a new array \( B \) of length \( m \);
  for \( k \text{ from } 1 \text{ to } m \text{ do} \)
    \textbf{repeat}
      Select random \( j \) such that \( j < i \);
      if \( \text{Pick edge} \) then
        Select random \( e \) such that \( 1 \leq e \leq m \);
        let \( c = A[j][e] \);
      else
        let \( c = j \)
      \textbf{end}
    \textbf{until} \( c \notin B \);
    append \( c \) to \( B \)
  \textbf{end}
  append \( B \) to \( A \);
\textbf{end}
\end{verbatim}

7.2.1 Parallelizing The Algorithm

If we wanted to parallelize this algorithm in such a way that we could add more than one node at a time, things becomes a bit complicated. This algorithm

\textsuperscript{18}All arrays here are indexed from 1.
is hard to parallelize since we have many dependencies between nodes. If we keep adding new nodes in parallel a newly added node might want to get the edge from a slightly older node that has not yet been assigned its new connection. If we were to pick another node at random whenever a node requests an edge from a new node that has not yet gotten its new connection it would skew the distribution. The likelihood of picking an older node would be disproportionately high since the older a node is the less likely it is that it would not be waiting to be assigned an edge and this would create an unwanted bias towards the older nodes. In the case that a new node is pending on a node that has not yet been assigned we would need to block on that node.

Things become even more complicated when we want to assign multiple edges to every node. Not only may the new node have to block on every node it wants to connect to but since we do not know which nodes the old nodes will connect to there is a chance that some of the nodes we are waiting for will be assigned the same connection and then we would have double edges. Therefore we would need to wait until all of the connections have been assigned and then check if there are any duplicates. If there are duplicates we would need to repeat the whole process over again. So a node that wants a connection from an older node would have to block until that node had gotten all of its connections and all of these connections are unique.

7.3 Using The Vat

We will now briefly discuss our attempt to solve this problem using Vats in Encore. For our implementation we have used an array to store the edges of the graph and we keep track of the indexes of the nodes by their corresponding index in the array. We let this array be an array of Single Valued Vats and whenever a node wants to get the edge of a node it uses the readOrRegister on that Vat and if the value is present it writes that value to itself and if the Vat is unassigned we register a post action that when triggered will write the value stored in that Vat to the current node.

The code for the Single Valued Vat solution can be found in Appendix A.1.

7.4 Using Multi-Valued Vats

To solve this problem for the case where each new node can get multiple connections we need to use the Multi-Valued Vat. We use And-Vats so that we can ensure atomicity in the sense that whenever a node gets assigned a new set of edges we know that all of these edges are unique. We let the array representing the graph be a list of And-Vats with as many fields as we want connections per node. As with the Vat solution we register a post condition on the nodes that we are pending on if they have not yet been assigned. But now we also need to have a pre and a fail condition. We need to have to
pro action in order to ensure that each connection is unique. We also need to introduce a fail action that, in the event that not all edges where unique, reselects a new set of candidate nodes and begins the process anew.

The code for the And-Vat solution can be found in Appendix A.2.

8 Discussion

In this section we will discuss what we have learned about the Vat from implementing it, what it has been like implementing this in Encore and examine the Vat with regards to some known properties desired in reactive-programming.

8.1 Implementing Vats

The biggest hurdle in implementing the vat has been choosing how to resolve the semantical issues discovered during the implementation of the Vat and during the implementation of the preferential-attachment problem. The solution for most of these issues and ambiguities were not apparent and for the most cases no perfect solution could be found and it went down to making design choices based on, what at the time, seemed to be the best solution. Further investigation into Vats may provide better solutions to these issues. Apart from this the actual implementation of Vats was straight forward. From our experience it seems likely that Vats could be implemented in any language supporting asynchronous callbacks and higher order functions.

8.2 Working With Encore

Encore delivers on its claim to enable parallelism by-default. The only part of the implementation of the Vat not made parallel is the execution of the actions, although this was more of a design choice due to limited support for task in Encore at the time of writing and not a fundamental shortcoming in the Encore language.

The fact that Encore is currently only in its development phase has been a constant reminder throughout the project. One reoccurring issue was the lack of correct parse errors and subtle inconsistencies in the language grammar. The fact that many parse errors do not indicate the actual error has made the debugging of simple syntactical errors very time consuming. The parse errors have a tendency to indicate an error unrelated to the actual problem and at a location in the code not related to the location of the actual error. In the case of subtle syntax errors or quirks in the implementation of the parser finding the actual error becomes very time consuming since it is often hard to find the cause of the error through visually inspecting the code. An example of this
kind of troubles is that the following piece of code `foo[0] ! bar()`, in which
the method `bar` is called on the first element in the array `foo` with an one-way
message send, compiles but if we change the one-way message call to a normal
method call like this: `foo [0], bar()` the program no longer compiles\(^{19}\) and gives a
parse error that is unrelated to the actual error. The reason that this does not
compile is due to bug in the operator precedence and the expression has to be
rewritten as `(foo [0]). bar()`. The fact that it is not possible to tell the reason for
the error from the error message or get a hint by the location of the error makes
problems like these, where the reason for the error cannot be deduced by simply
inspecting the code is extremely difficult. Another consequence of Encore being
in the development stage has been that there have been bugs present in the
Encore run-time. These have made the debugging of programs difficult since in
many cases it has not been apparent if the reason for the program failing has
been due to a bug in the Encore code or a bug in the run-time. This has led
to debugging often having to be done from two different directions and as we
all know, every time you try to fix a bug to run the risk of introducing another
one. This has in practice lead to a lot of time being spent trying to find and fix
non-existing bugs without actually being close to fixing the actual problem. In
many cases these problems only arise intermittently and in more complicated
programs, making the bug harder to isolate and thus harder to fix.

Another prevalent source of problems has been that the Encore garbage collector
has at the time of writing been inherently unsafe this is because Encore is based
on the Pony run-time and uses the Pony garbage collection. The Pony garbage
collector is based on the ORCA \cite{7} protocol to support concurrent garbage
collection across different actors. But the ORCA protocol requires there to be
restrictions on how active-objects can modify passive objects shared between
active-objects. These restrictions could be enforced with a type system that
restricts what one can do with references, such as a ownership type system or
reference capabilities. Without such a type system it is up to the programmer
to make sure that any shared passive objects are handled in the correct fashion
and this can be very hard to do in some situations.

Encore as a language has been very useful and there have been no fundamental
shortcomings in Encore that has complicated the implementation. Modulo bugs
Encore has been a very good language choice for implementing the vat.

### 8.3 The Power Of Vats

In this section we briefly discuss our experience of applying Vats to the prefer-
tential attachment problem and also compare Vats against some previously
studied desirable properties of reactive-programming constructs and models.

\(^{19}\text{At the time of writing this}\)
8.3.1 Vats And The Preferential Attachment Problem

Using Vats to solve the preferential-attachment problem has been a very smooth process. The simplicity and inherent concurrency of the Single Valued Vat made the simple case of the preferential attachment problem\(^{20}\) was very trivial to implement. The only thing that one had to was to add a single post action in the case that a node needed to wait for an older node to get its connection assigned. We consider the solution of the simple case of the preferential attachment problem to be only about 50 lines of Encore code to be a good indication of the Single Valued Vats usability and simplicity.

For the harder preferential attachment problem\(^{21}\) the solution is more verbose and not as clean as the solution for the simpler problem. Although considering the fact that the general case of the preferential attachment problem is much more complicated than the trivial one the solution using And-Vats is not overly complicated. This has been a good case to see how the different types of actions in the Vat can be used to work together. Using the pre and fail actions provided a smooth way to ensure that there where no double connections. This in conjunction with the And-Vats inherent atomicity has led to the handling of the rollbacks, required when there is a double edge, to be resolved with very little effort.

8.3.2 Taxonomy Of The Vat

We will now perform a brief examination of the Single Valued Vat and Chocolate Vat with respect to the taxonomy\(^{22}\) of reactive programming put forth by Bainomugisha et al [1].

8.3.2.1 Basic Abstractions As of now it would appear that Vats would be classified as propagators in their current implementation as they require values to be explicitly written by the programmer but we have found no indication that Vats could not be used to encode events if there where to be support in the underlying implementation to support automatic writes to the Vats and thus make them suitable for handling event streams. No conclusions can as of yet be drawn regarding the vats ability to handle continuous time varying values.

8.3.2.2 Evaluation Model The Vats inherently support both push and pull-based. The post and fail actions provide push-based behavior since the entity that consumes the data will need to do no explicit requesting of the data in the

\(^{20}\) Where each new node gets only one new connection.

\(^{21}\) In which new nodes can get an arbitrary (yet fixed at the start of the program) number of new edges.

\(^{22}\) Described in Section 4.1.
Vat. The ability to explicitly read from the Vat provide pull-based behavior since an entity can at its own discretion decide when to request to read the value stored in a Vat. The Vat enable an entity to use both of these evaluation models simultaneously and add pull based subscriptions through the actions and explicitly request the value when needed.

8.3.2.3 Glitch Avoidance The flavors of the Vat examined in this work does not provide support for the avoidance of glitches in general in the form described by Bainomugisha et al. Although the And-Vat can be used to ensure that synchronicity is provided in some situations. And-Vats fails to provide protections of glitches in this sense to situations where the values that must be kept in sync at certain points can not be co-located, i.e they can not reside in the same And-Vat. Although the Nested Vat and Linked Vat may provide general protection against glitches. There exists the possibility that Linked Vats could provide protection against glitches in a distributed setting.

8.3.2.4 Lifting Only manual lifting can be achieved with the current implementation of the Vat. For a language that does not support operator overloading, lifting of Vats would require the Vat to be implemented natively into the language and the languages type system may have to be extended to support lifting of Vats. As of now there are no plans of implementing the Vat natively in Encore. This may become desirable if further evaluation of the Vat shows the Vat to be powerful enough to warrant the effort needed to incorporate the Vat into the backend of Encore. Although a native implementation of the vat could solve many of the shortcomings of the current implementation.

8.3.2.5 Multidirectionality As with the example provided by Bainomugisha et al. where we have a system in which there are two producers that give the temperature of the same source but one gets the value in Celsius and the other in Fahrenheit, we would like to enable the behavior that whenever one of the two producers are updated the other will be updated with the same temperature in its corresponding unit.

For a simple case where say only the Fahrenheit temperature changes on updates to the Celsius temperature this can be easily achieved using Single Valued Vats where the post action of the Vat containing the Celsius value writes to the Vat containing the Fahrenheit temperature.

The case where both the Celsius temperature and the Fahrenheit temperature updates each other is not possible to achieve using regular Vats by simply having two Vats that update each other using their respective post actions. This introduces feedback loops, where both Vats keeps updating each other constantly, this can not even be solved through adding a pre action that rejects writes that do not change the value, since a sequence of consecutive writes
of different values to one of the Vats because this can cause the writes to be interleaved in such a way that the pre actions never catch the write that was send through the feedback. Or-Vats can solve this problem through its transformation actions but this forces the two values to be co-located. Nested Vats with Or semantics could solve this problem.

8.3.2.6 Support For Distribution As of now no definitive statements can be made about the Vats possibility to support distributed programs. Linked Vats and Nested Vats may provide some support for distribution due to their lack of dependency on co-location, although not enough is known about the Linked Vat and Nested Vat to make any claims.

8.3.3 The Vat Like Constructs

Not much can be said about the Vats relation to the other concurrency constructs described in Section 4.2 at this point. But at first glance it appears that that the Vat is somewhat more powerful than the other related constructs due to the Vats more complicated system of actions. But more work is needed before any definite conclusions can be drawn.

9 Future Work

In the previous section we concluded that Vats could support many of the basic properties for reactive programming and potentially contribute to the open problem of distributed reactive programming as put forth by Bainomugish et al. But in order to draw any definitive about the Vats role in reactive programming a lot of work remains to be done. The Vat should also be compared against other known concurrency constructs similar to Vats. Although this also needs further evaluation and experimentation of Vats.

The And-Vat and Or-Vat are not the only Chocolate Vats presented in the original paper by Clarke and Wrigstad [6]. There are also nested Vats, linked Vats and type changing Vats. These would also need to be implemented and evaluated. Implementing them would require contemplation regarding the various design choices that needs to be made and how to overcome possible issues and semantic ambiguities that may arise when implementing these Chocolate Vats. These Chocolate Vats may have applications to the issues put forth by Bainomugish et al. These more exotic Chocolate Vats would open up a slew of new possible problems to apply Vats to and see whether or not Vats construct provides an easy and flexible tool for reactive-programming. One should also perform a comparative study of solutions for these applications using Vats and other well known methodologies from reactive-programming.
After all of the flavors of Vats has been implemented and studied the semantics of the Vats needs to be formalized. In order to enable us to make any definitive statements regarding the expressiveness of the vat and its applications to the open problems in reactive-programming.

10 Conclusion

After this thesis we have a working initial implementation of the elementary Vats. I have also put Vats through an initial test and found that vats can be used to solve some real problem. When applying the Vats to the problem of generating scale-free graphs I found them to be easy to use and that they provided a very simple and intuitive solution to an otherwise complicated problem.

The implementation of the vat went very well and I found no need to do any elaborate hacks or use very specialized concurrency constructs. Therefore I see no reason for why the elementary Vats could not be easily implemented as easily in many other languages.

Only time will tell if the design choices that has been made are the correct ones, but as of now we have an implementation and an understanding of how the vats work that is sufficiently strong to enable people who want to experiment with Vats to do so.

I feel that we now have a foundation for the Vats strong enough to continue exploring and testing the vats.

References

A Preferential Attachment Code

In this appendix you will find the code for the implementation of the preferential attachment problem.
A.1 Using Single Valued Vats

```python
import Vat
import Random

class Main
    def main(args : [String]) : void
        let nodes = 0;
        match (args[1]).to_int() with
            Just n => nodes = n
            Nothing => {print "please provide the number of nodes."; exit(-1)};
        print("Nodes: {}\n",nodes);
        randomSeed(getUTime());
        let graph = new Graph(nodes);
        graph.run();
    }

class Graph
    vat_array : [Vat<int>]
    def init(size : int) : void {
        this.vat_array = new [Vat<int>](size);
        for index in [0 .. |this.vat_array|-1]{
            this.vat_array[index] = new Vat<int>();
        }
    }
    def run() : void {
        this! updateEdge(1,0); --Initialize the first node
        for index in [2 .. |this.vat_array| - 1]{
            let candidate = randint(1,index -1 ); --Pick older node
            if(randomBool()) then {
                this! updateEdge(index,candidate); --Just connect to that node
            }else{
                let post_action =
                    \(old: Maybe\ int, new_val: int) ->
                    (this! updateEdge(index,new_val); false);
                -- Attempt to get the edge and if no edge has been assigned
                -- register a new post action.
                (this.vat_array[candidate]).readOrRegPost(post_action) -->
                    \(ret: (bool, Maybe\ int)) -->(
                        match ret with
                            (ignore, Just x) => this! updateEdge(index,x)
                            _ => ()
                    )
            }
        }
    }
```
def updateEdge(node : int, connection : int) : void {
    (this.vat_array[node]).write(connection) --> \(x : bool\) ->
    print("{} -> {}\n",node,connection);
}

A.2 Using And-Vats

import AndVat
import Random

class Main
    def main(args: [String]) : void{
        let seed = getUTime();
        randomSeed(seed);
        if(|args| < 3) then {
            print "Arguments are number of nodes and number of connections";
            embed void abort();
        }else{
            let nodes = match (args[1]).to_int() with Just x => x Nothing => -1;
            let connections = match (args[2]).to_int() with Just x => x Nothing => -1;
            print("Nodes {}, Conenctions {}
",nodes,connections);
            let graph = new Graph(nodes,connections);
            get graph.dumb();
            graph!run()
        }
    }

class Graph
    vat_array : [AndVat<int>]
    connections : int

    def init(nodes: int,connections: int) : void {
        this.connections = connections;
        this.vat_array = new [AndVat<int>](nodes);
        repeat i <- nodes{
            this.vat_array[i] = new AndVat<int>(connections,false,Nothing)
        };
        for i in [(this.connections) .. |this.vat_array|-1] {
            let vat = this.vat_array[i];

            -- Post action that just prints the edges.
            vat ! registerPost(\(old: Maybe [int],edges: [int]\)) ->
            {this!printArray(i,this.clone(edges));false}
        }

        -- Persistent pre action to enforce that all connections are unique
        vat ! registerPre(\(old: Maybe [int], edges: [int]\)) -> {
            let not_eql = true;
            repeat j <- |edges| {
                repeat k <- |edges| {
                    if(k != j) then (not_eql = not_eql and edges[k] != edges[j])
                }
            }
        };
}
(true, not_eq)
);

-- Persistent fail that attempts to pick new nodes in the case
-- of conflicts.
vat ! registerFail((old: Maybe [int], edges: [int]) ->
{ print "failing"; this ! pickNew(i); true}
);

}

def block() : void {} -- Encore workaround.

def run() : void {
  repeat i <- this.connections {
    --Initialize the first node
    this.vat_array[this.connections] ! singleWrite(i,i);
  }
  for i in [this.connections+1 .. |this.vat_array|-1]{
    this!pickNew(i);
  }

  --This methods picks the new candidate edges for a node.
  def pickNew(index: int) : void{
    repeat edge <- this.connections {
      let node = randomInt(this.connections,index-1);
      if randomBool() then {
        this.vat_array[index] ! singleWrite(edge,node) --Just connect to the node
      } else{
        -- Write new edge if it is present at the selected node,
        -- otherwise just add the post action.
        (this.vat_array[node]).readOrRegisterPost((old: Maybe [int], edges: [int]) ->
        (this.vat_array[index] ! singleWrite(edge,edges[edge]); false)
      } -> (node_edges: Maybe [int]) ->{
        match node_edges with
        Nothing => ()
        Just e => this.vat_array[index] ! singleWrite(edge,e[edge])
      }
    }
  }

  def printArray(index: int, arr: [int]) : void{
    for edge in arr{
      print("{} -> {}
    ,index,edge);
    }
  }