Bachelor Informatica

University of Amsterdam

Purely event-driven programming

A programming language design

Bas van den Heuvel

10343725

Wednesday 8th June, 2016

Supervisor(s): Dr. ir. B. Diertens and Dr. A. Ponse

Signed: Signees

Abstract

Purely event-driven programming is a style of programming in which the control flow is solely organized by events. This thesis describes a programming language design for purely event-driven programming. The language is designed to have implicit concurrency without inversion of control. It consists of state machines that communicate through events. State machine code is executed and scheduled by a system called MachineControl, which also manages events. Experimentation has shown that the language is suitable for algorithms that involve workers and for naturally event-driven systems. The language is less suitable for algorithms that are sequential in nature, but it can still be used as a general purpose programming language.

Contents

CHAPTER 1

Introduction

What would a programming language without function calls, explicit concurrency and/or a sequential control flow look like? Such a programming language would be *purely event-driven*. Can it support concurrent event-driven programming without inversion of control? This thesis proposes such a language, with experiments to answer these questions.

What is purely event-driven programming? To answer this question, another question needs to be answered: what is event-driven programming? Event-driven programming is a *programming paradigm*, but there are many ways this paradigm is interpreted. One programmer could call a program event-driven, while another would completely disagree. So, to give a more theoretically founded answer to this question, one more question needs to be answered first: what is an event?

An event is essentially *something that happens*. This can be a leaf falling from a tree, but it can also be a concious decision being made. So, anything happening is actually an event.

An event-driven system has a control flow which is determined by events. So, event-driven programming is a way of programming in which certain actions cause events and vice versa. As an example, event-driven programming is widely used in User Interface (UI) programming. A user interacting with the UI (e.g. the user clicks a button) is an event, but a window being ready is also an event. When working with a certain system, not all events are relevant. Irrelevant events can therefore be ignored as events.

Event-driven programming is mostly supported as an addition to classical sequential programming. In these systems the programmer attaches functions (event handlers or callbacks) to certain events. As opposed to having to stop a program to wait for an event to happen, the event system will execute the registered function once the event occurs.

In [1], this programming model is called *inversion of control*, because the event system takes a program's control flow away from the programmer. Inversion of control is problematic, because the fragmentation of program logic makes a program's flow difficult to understand.

In [1] and [2], it is claimed that event-driven programming is suitable for concurrent programming, but event-driven programming is difficult because of inversion of control. The mixture of sequential programming and events can make programs even more difficult to understand.

In Chapter 2 on page 9, a language design for purely event-driven programming is proposed. This section describes the thesis' definitive language design. In Chapter 3 on page 17 experiments are described that have changed the language. This section explains problems with earlier design choices, leading to this definitive design. The thesis is concluded with proposals for further research and a conclusion.

CHAPTER 2

A purely event-driven programming language design

Following is a description of a purely event-driven programming language design. Step by step, several concepts and mechanisms are introduced, building up to a complete programming language. These descriptions are accompanied by code examples. The syntax in these snippets is just an option out of infinitely many possibilities, merely chosen to make the concepts easier to understand. It is loosely based on the syntax of C. These syntax choices are not explained, because syntax is not the topic of this thesis.

This language is designed with concurrency in mind, which is explained in Section 2.4 on page 15. The description of the language does not include an implementation, but it can be used when developing one. However, as described in the introduction, this concurrency should be implicit.

The choice made for concurrency in this language design is *state machines*, often simply referred to as *machines*. They are the building blocks of the language. Machines describe some behaviour, such as a computation or I/O, but they do not execute it. State machines are completely independent, so they can be represented on a computer as an independent unit (e.g. a process or a thread). The way state machine code can be executed also supports concurrency, as it requires little overhead. This will be explained in following sections.

Machines can communicate through *events*, combining to form programs. State machines and events are managed by a system called Machine Control (MC), which is also responsible for the execution of state machine code.

Two diagrams are included with this language description: state machine in Fig. 2.1 and MC in Fig. 2.2. Every part will be explained throughout this section, so they can be used as a visual guidance.

2.1 State machines

State machines represent units that perform specific tasks. They communicate through events, so they do not need to be aware of each others' behaviour and contents. This makes state machines suitable for concurrent execution. Therefore, they are designed to maintain this independence.

State machines have space for *local variables*. A state machine comprises an infinite number of states and is always in such a *state*. Machines can switch between states through *state transitions*. State machines only describe a behaviour, they do not execute it.

2.1.1 Local variables and arguments

A state machine is defined with an arbitrary amount of local variables. These can be accessed from any state in the machine, but other machines can not reach them. This keeps the space a machine needs constant, with the possible exception of data structures such as arrays. Arrays

might might vary in size throughout the existence of a machine. How these exceptions are handled is up to the implementation.

These local variables are declared for an entire machine. It is not possible to declare new variables within states. This makes the scope of variables unambiguous. It also limits the amount of bookkeeping necessary for concurrent execution.

Besides local variables, machines can require arguments. Aside from their values being assigned with the instantiation of a state machine, arguments are identical to local variables.

To maintain independence of machines, implementations should consider how to pass argument values other than constants, especially when passing references to objects and the like. Using a reference to an object would require a machine to reach the memory of the object's creating state machine.

2.1.2 States

In essence, a state is a piece of code. Code in a state consists of actions on variables, if-statements and loop constructions. As the goal of this paper is to study event-driven programming as a concept, more detail on these actions is not given. In example, provided state code is pseudo code.

The language adds some functionality that can be used in states: state transitions and MC and event statements. These are described in the following sections. On the other hand, return statements do not exist in this language.

2.1.3 State transitions

State transitions are like return statements. They stop the execution of a state. The difference is that a state transition merely changes a machine's state, i.e. what piece of code to run. A return statement involves much more bookkeeping, such as returning control to a context and passing values.

If the execution of a state's code does not reach a state transition, the machine implicitly transitions to the *listen state*. This state is very important as it handles a machine's incoming events, as explained in Section 2.3.4 on page 13.

In example code, a state transition to state a looks as follows: `=> a`.

2.1.4 Special states

There are three *special states*: the *initial* state, the *listen* state and the *halt* state. The initial state is an indicator which tells the machine in which state to start. The listen state handles events. It is implicit, meaning that it exists, but its functionality is provided by the language's

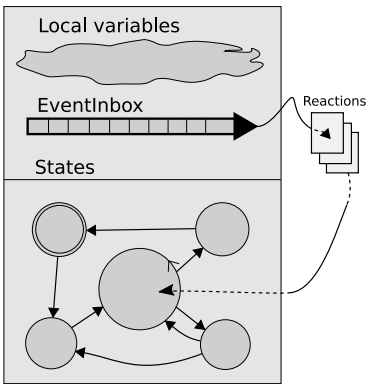

Figure 2.1: A diagram of a state machine, showing local variables, the event inbox, states and the path of events from inbox through reactions to listen state.

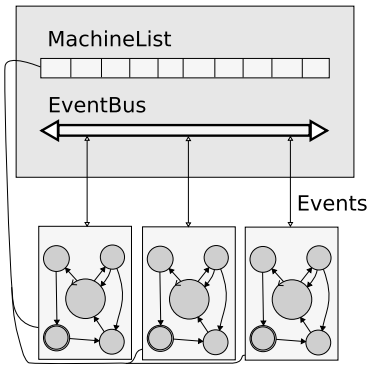

Figure 2.2: A diagram of MC, showing the machine list, event buss and communication with state machines.

execution system. The halt state is another implicit state which makes the machine's execution stop. All these states are explained more elaborately in following sections.

2.1.5 An example state machine

A very simple state machine, as described above looks as follows. The declaration `init state a {...}` indicates *a* as the machine's initial state. The machine iterates `i` from 0 through `n` and then halts.

```
machine machine_1(n) {
    i = 0;

    init state state_1 {
        if (i < n) {
            // do something.
            i += 1;
            => state_1;
        }

        => halt;
    }
}
```

2.2 MachineControl

State machines only describe behaviour. To actually execute it, the language has a system called MachineControl (MC). It does all the bookkeeping for execution of states and communication through events. The management of events is a system in its own. Therefore, it is not explained here, but in Section 2.3.

MC keeps track of all active state machines in its *MachineList*. The execution of a machine's state is called *cycling*. MC is always present in a program, so it can be accessed from any state. In example code, MC is referred to by the constant `ctl`.

2.2.1 Cycling

The execution of the code in a machine's state is called a cycle. The result of a cycle is the state in which the machine results after the cycle, determined by a state transition. In order to run a program, MC needs to make sure all machines in its MachineList get to cycle.

All machines have to be cycled. If any are left out, the result of a program could be influenced. The program will not be able to halt, since a machine only halts after cycling its halt state. So, some sort of scheduling needs to take place in order to cycle all machines.

The simplest way of scheduling is to treat the MachineList as a queue, cycling the first machine and placing it in the back of the queue. Scheduling can be more advanced, for example through analysing the duration of cycles and prioritising faster states. Anything is possible as long as eventually all machines get cycled.

2.2.2 Starting a machine

To start a machine, this needs to be announced to MC. MC instantiates the machine by creating space for it, initializing local variables and putting it in its initial state. The new machine is added to MC's MachineList. Finally, MC passes the new machine's identity as a reference to the requesting machine. This reference can be used for event emissions, as explained in following sections.

For example, when machine *a* wants to instantiate machine *b* and store the reference in local variable `machine_b`, it should have the following code in a state:
`machine_b = ctl.start(b, arguments)`. Remember that the variable used to store the reference has to be defined as a local variable before being able to assign anything to it.

Every machine has a *context*, which is the machine instantiating it. This is useful for events, as explained in section 2.3.3 on page 13.

2.2.3 Starting (and stopping) a program

A program will not run unless at least one machine is started. This can be done by invoking MC from outside state machines. Only a state machine and arguments are needed. It is, however, possible that an implementation has additional arguments, such as a debug flag. Running state machine *a* can be done by putting the following statement at the bottom of a program: `ctl.run(a, arguments)`.

This statement causes MC to instantiate machine *a*, as it would when called from a state. This means that this first machine *a* will get cycled, thus running the program. Once all machines in MC's MachineList are in their halt state, MC is done running. At this point MC will be reset, as described in Section 2.3.7 on page 15. It is possible to have multiple sequential run statements. They will be run in the given order, after which the program is finished.

The first running machine has a context, like all state machines. It is, however, different from other machines, since there is no machine starting it. The context of this machine is an empty non-active state machine. The only reason for this is generalisation: when implementing this language the first machine should be no different from other machines.

2.2.4 Stopping machines

A state machine's execution stops when it reaches its halt state. This implicit state notifies MC about the machine halting, which removes the machine from MC's list. Depending on the implementation, this is also the time for any garbage collection, as described in Section 2.3.7 on page 15.

2.3 Events

At this point, the description of the language design comprises state machines and MC, a mechanism to run them. However, the most important part is still missing: events. Events are used for communication between machines. An event describes something that has happened.

An event is a form of message passing, with the important aspect that events do not necessarily have a destination. State machines create events by *emitting* them, but they are not responsible for delivering them to other state machines. Thus, from the state machine's point of view, an event is just an event, i.e. a notice that something happened. State machines are, however, responsible for reacting to events. All event passing happens through MC, which takes care of distribution.

An event contains two fields: a *type*, which is a string, and an *emitter*, which identifies the machine that created the event. Events can also hold an optional *value*, which can be anything, and an optional *destination*, which tells MC where to deliver the event to.

2.3.1 Event emission

MC holds a buffer, called the *EventBus*. When a machine announces an event to MC, the event is added to the EventBus. From there, MC will distribute the event to state machines, as explained in the next section.

Event emission is done through the *emit* statement. The event's emitter is implicit.

```
emit ("type");         // event without value and destination
emit ("type", value);  // event with value, without destination
emit (...) to machine; // event with destination
```

2.3.2 Event distribution

MC distributes events in its EventBus to all active machines. State machines have a queue, the *EventInbox*, into which MC puts events. Events never get distributed to the EventInbox of their emitter. If an event has a destination, it will only be distributed to that destination machine (if it is still cycling, i.e. not in its halt state).

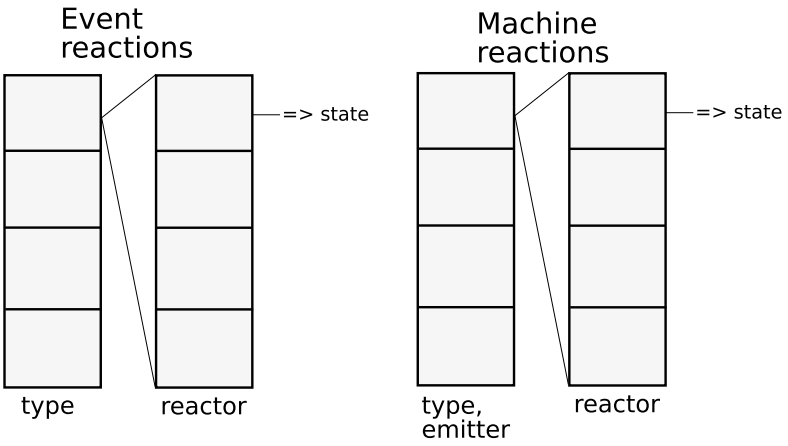

Figure 2.3: Both reaction maps as held by MC.

Like with state machine cycling, MC is responsible for scheduling this distribution. The language has been designed to be ignorant of this scheduling, which is why MC does not take care of actual reactions. Reactions are handled by machines, as described in following sections. Scheduling can be anything, such as simply distributing all events before a cycle, or maintaining an independent event distribution process.

2.3.3 Event reaction

Purely event-driven communication starts with event emission and ends with a reaction. A machine can assign a reaction to a type of event in the form of a state transition. The combination of an event type and a state transition is called an *EventReaction*. It is also possible to make the reaction more specific by requiring an emitter (i.e. the emitter's identifier), which is called a *MachineReaction*. A reacting machine is called the *reactor*.

MC contains two two-dimensional maps, one for EventReactions and one for MachineReactions, as in Fig. 2.3. A machine can register a reaction to MC through the *when* statements. This stores the given state transition in either map, first under the event's type (combined with its emitter in case of a MachineReaction), then under the reactor's identifier. If there is already a reaction for a certain type, the existing state transition gets overwritten with the new one. The actual reaction takes place in the listen state.

Different states can require a machine to stop reacting to certain events in the future. This can be done through the *ignore* statements, which remove the entry from either map.

```
// Register reaction.
when "type" => state;
when machine emits "type" => state;

// Ignore reaction.
ignore when "type";
ignore when machine emits "type";
```

This is where a machine's context comes in handy. It allows a machine to react to some signal from its context, without being aware of what the context actually is. This is an important aspect of the programming language which supports abstractions and generalised code.

2.3.4 Listen state

A machine's listen state is where the actual reaction happens. Once a machine enters the listen state it will check the EventInbox for events. If there are events, the first one is taken from the queue. Using MC's reaction maps the machine will decide what to do with the event. Because MachineReactions are more specific than EventReactions, these will be checked first. If a reaction entry is found, the entries' state transition will be performed. There is something else that

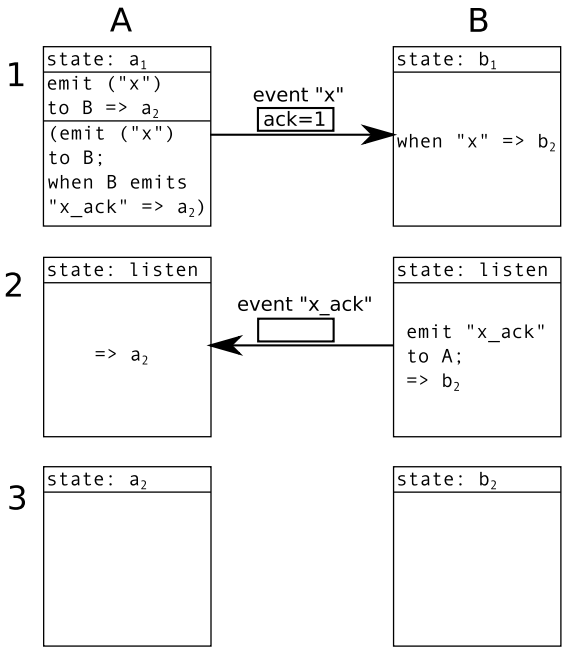

Figure 2.4: Machine A emits an event "x" to machine B, which acknowledges.
Step 1: A emits event "x" with acknowledgment flag and assigns reaction to acknowledgement. B assigns reaction to event "x".
Step 2: B emits acknowledgement event "x_ack" to A and transitions to b_2. A receives acknowledgement and transitions to a_2.
Step 3: machines are synchronised.

happens before the state transition is taken. This has to do with event acknowledgements, which will be explained in a following section.

If there are no reactions assigned to the event, the machine will dump it and re-enter its listen state. If there are no events in the EventInbox, the listen state will be re-entered as well.

Events play a central role in this programming language, which makes the listen state very important and frequently used. This is the reason why an explicit state transition to listen is optional. When a state's code has been executed, but no state transition occurred, there will be an implicit transition to the listen state.

2.3.5 Halt reaction

Before halting, a machine needs to make all machines it started halt. Therefore, all machines get instantiated with one initial reaction: `when ctx emits "halt" => halt`. Before announcing its halt to MC, the machine emits an event, typed 'halt'. So, a machine halting, will halt all machines it started implicitly.

2.3.6 Event acknowledgements

Because of the concurrent nature of the language, machines in a program will frequently be required to synchronise. In the context of this programming language, this means that machines need to synchronise event handling. Suppose for example a state machine A that uses multiple instances of a state machine B for a computation. Some B machines are done more quickly than others. So, before A can continue it has to synchronise with all B machines.

Synchronisation could be necessary because of the nature of the program (e.g. machines need to run at the same moment) or because it will simplify the program (as in Section 3.4.1 on page 22). This can be done through event acknowledgements, explained visually in Fig. 2.4.

An event acknowledgment means that the reactor *acknowledges* a reaction taking place. This is done by emitting a special event. The machine that sent the original event will receive this

acknowledgment. This machine will react to this special event with a state transition, provided with the emit statement: `emit ("type") to machine => state`.

This statement automatically sets up acknowledgment handling. The event gets an acknowledgement flag, which tells the receiving state machine to acknowledge its reactions. In a machine's listen state, before performing a state transition, this flag is checked. If necessary, the reacting machine will emit an event. An event type is automatically generated. If an event's type is "type", the acknowledgement type will be "type_ack".

The emitter of the original event gets a reaction assigned to this acknowledgement with the given state transition. This way, once the emitter is in its listen state and the acknowledgement has been received, the machines become synchronised.

2.3.7 Garbage collection

Once a state machine is halted, all the space it takes is waste. Therefore, when a machine halts, all this space needs to be wiped. All local variables have to be cleaned, including any references to these variables.

The machine's EventInbox needs to be deleted, as well as any events the machine has emitted. This means that when a machine halts, all leftover events in any machine's EventInbox and in MC's EventBus should be removed. Also, any reactions for this machine in MC should be deleted. Finally, the space for the machine can be freed.

This process is completely dependent on the implementation. Were one to implement this language in a language (i.e. simulate) with proper garbage collection, almost nothing has to be done. A native implementation, however, would require freeing all the memory explicitly. The latter would be the language's actual garbage collection.

2.4 Concurrency

Because of the communication through events, state machines are independent units. They do not need to access each others' memory. This takes away the need to make every interaction between state machines safe for concurrency. Communication happens through the independent MC, which takes care of safe interaction with machines.

Another advantage of using state machines and events is the elimination of a call stack. A call stack is necessary for one method to invoke another method. It provides a context, takes care of results being returned and makes the program continue with the right code after the called method is done.

In [3] this behaviour is summarized as three components: coordination, continuation and context. Coordination makes sure the calling method waits for the called method to finish. Continuation takes care of continuing with the following code. Context is the collection of variables to which the called method should have access. The context aspect takes care of restoring the proper variables once the called method is finished.

State machines and events eliminate all three of these components. Coordination *requires* a program to have a specific order. State machines are independent units that can do their work without having to wait for other machines to deliver results. This is, however, still possible through synchronisation (i.e. event acknowledgments).

This also eliminates continuation. A machine does not need to await an event emission or reaction to finish, so there is no need to store a continuation on some stack.

Finally, a machine's context never changes. All local variables are declared upon instantiation and they remain throughout the existence of the machine. Accessing variables from other machines is done through events, so a machine will never need another machine's context.

CHAPTER 3

Experimentation

What does it mean to test a programming language design? What it does not mean is to fully implement a compiler or interpreter. When programming for such a working implementation results in the discovery of design flaws, the whole process has to be started over. It does mean to manually test programs in the language design, pointing out the design's flaws, providing immediate feedback.

3.1 Designing programs

Purely event-driven programming is different from classical programming, in that a program's structure is spread across state machines. In turn, the control flow inside such machines is spread across states and interaction is moved to events. This makes designing programs from scratch rather difficult.

Instead of trying to write a program directly in the desired syntax, it is wise to graphically draw the program. State machines are very suited for this. They consist of states, which usually are (named) circles in drawings. Transition between the states are indicated by arrows, often labeled by a condition. A state can contain some text explaining actions it performs.

To design a program, one can start with a crude graph, showing the general flow of the program. This design can then be refined with more states and more meaningful actions. This means that a design might need several state machine drawings before one can actually start programming.

Once a design is finished, it is trivial to implement it in the state machine programming language described above. This step is not even necessary for experimenting with the language design, since the syntax is of no matter.

3.1.1 Example: Sieve of Eratosthenes

A good example to show that purely event-driven programming is suitable for concurrent programming is the Sieve of Eratosthenes. It is an algorithm that finds prime numbers. The Sieve does so by starting with a number (usually the first prime number 2), marking all its multiples. The next unmarked number is the next prime number and the process can be repeated. The process is visually shown in Figures 3.1 and 3.2.

Figure 3.1: Sieve of Eratosthenes, first iteration [4].

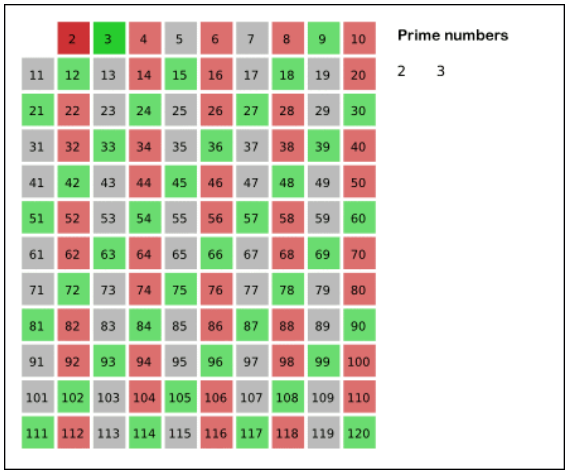

Figure 3.2: Sieve of Eratosthenes, second iteration [4].

This example uses a slightly different algorithm. Instead of marking off multiples of primes, a new prime number spawns a *picker*. This picker has a counter, starting at 0. Each time a new number is tried, the counters of these sieves is incremented. If a sieve's counter equals the prime number that spawned it, the new number is a multiple of a prime number. So, the counter is reset and the next number can be tested. In Python this algorithm looks as follows:

```
def sieve(n):
    pickers = []
    x = 2

    while n > 0:
        success = True

        for p in pickers:
            p['counter'] += 1
            if p['counter'] == p['prime']:
                p['counter'] = 0
                success = False

        if success:
            pickers.append({
                'prime': x,
                'counter': 0})
            print('Found prime', x)
            n -= 1

        x += 1
```

To make this algorithm purely event-driven a separation into state machines needs to be made. Looking at the code, it becomes apparent that the code can be split up into two state machines. One state machine is the sieve itself (called *Sieve*), incrementing x and spawning new pickers. The other is the picker (called *Picker*), incrementing its counter with each new value of x and determining success or not.

So, each time *Sieve* increments x, it should let the *Picker* machines run. It can do so by emitting a "run" event. The *Picker* machines will then increment their counter. Each of them will emit either a "fail" or a "pass" event. If the *Sieve* receives a "pass" event from all *Picker* machines, a prime is found. If any *Picker* emits "fail", *Sieve* can immediately start a new round of sieving.

However, all *Picker* machines need to increment their counter. Because of this, synchronisation is required, as explained in detail in Section 3.4.1 on page 22. This means that the *Sieve* will have to wait for all *Picker* machines to finish. For this, an acknowledgment is required for the "run" event.

These state machines can be drawn, but the drawing for *Sieve* becomes cluttered and complex due to the synchronisation. This problem can be limited by adding another state machine:

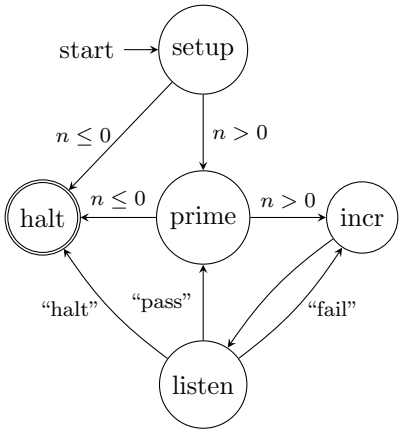

Figure 3.3: State machine *Sieve*. State workings: *setup* starts *PickerManager* and registers reactions to "pass" and "fail"; *prime* emits "new_prime" to *PickerManager*; *incr* increments x and emits "new_x" to *PickerManager*.

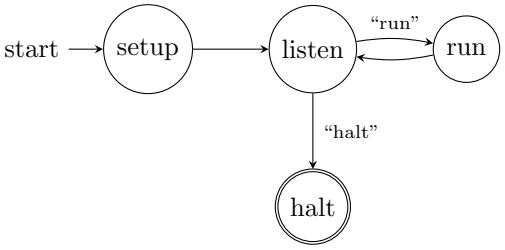

Figure 3.4: State machine *Picker*. State workings: *setup* registers a reaction to "run"; *run* increments counter and either emits "pass" or "fail".

PickerManager. This machine takes care of running and synchronising *Picker* machines. It also collects "fail" and "pass" events, passing them on to *Sieve* as a single result.

The resulting state machines are displayed in Figures 3.3, 3.4 and 3.5. Code that runs this implementation of the Sieve of Eratosthenes is included in Appendix A on page 41.

3.2 Testing programs

Testing the state machines can be done by keeping a list of active machines and their current states, whilst following the arrows in the graphical drawing and performing actions in states manually. This can lead to design errors being discovered. Such errors can easily be spotted in the graphical design, if the programmer has an understanding of their program's flow. By updating the graph and repeating this process, one can test and verify a program design.

3.3 Simulation in Python

Manual bookkeeping works for simple programs. However, once a program contains more machines and machines become more complex, testing becomes more and more error prone. Therefore, a simulation of the language in Python is provided with this thesis.

A simulation of a language means that its workings are modeled, in this case in another programming language. Such a programming language needs the possibility to create state machines, to run a program through some implementation of MC and all types of management described in this thesis.

Python's object-oriented approach is much suited for this. State machines can be implemented as classes, as will be explained. MC can be implemented as another class, containing all necessary bookkeeping, such as the MachineList and EventBus. Data structures such as lists and queues are readily available in Python.

The simulator contains three classes. The class `MachineControl` implements MC: scheduling of cycling, tracking state machines and distributing events. The class `Event` represents events. It is only used in the simulator internally, as the creation of objects of this class has been abstracted to methods of `StateMachine`. This class is the implementation of the most basic state machine,

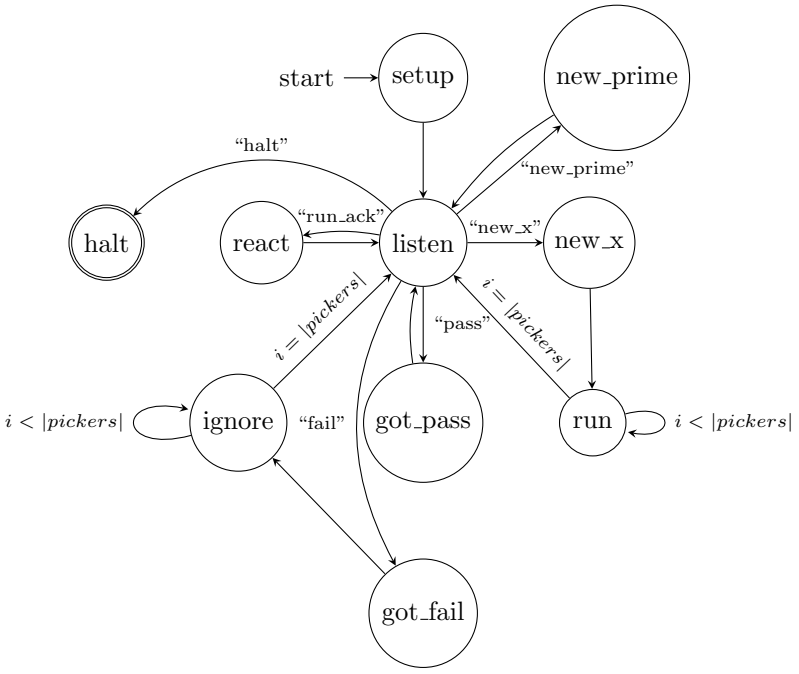

Figure 3.5: State machine *PickerManager*. State workings: *setup* registers reactions to "new_prime" and "new_x"; *new_prime* starts a *Picker*; *new_x* sets up a counter; *run* emits "run" to all *Picker* machines with acknowledgement; *react* is the acknowledgement state for "run", registers reaction to a *Picker*'s "pass" and "fail" events; *got_pass* increments counter and if enough, emits "pass" to *Picker*; *got_fail* emits "fail" to *Picker*; *ignore* ignores future "pass" and "fail" events from all *Picker* machines.

containing only a listen and halt state. It has several methods for working with events and MC. By creating a subclass of `StateMachine`, state machines can be created.

Cycle scheduling is implemented as a simple queue. Every cycle, the first machine gets cycled, after which it is placed in the back of the queue. When a machine halts, it is removed from this queue. Events are emitted through a method that places an `Event` object in `MachineControl`'s EventBus. MC distributes all events on the EventBus before each cycle.

The full documentation for the Python simulator can be found in Appendix B on page 45.

3.3.1 Writing programs

The syntax for writing programs in the Python simulator is different from the syntax in the language description in this thesis. Therefore, a translation is explained, containing examples. Note that in these Python examples there are occurrences of "\". This is only to make the code correct, while still fitting on the page.

Simulating programs is done by writing state machines as subclasses of `StateMachine`. Methods of this class represent states. They never contain any arguments, except `self`, which makes the method an *instance-method*, thus acting on the instance instead of the class. State transitions are done by returning one of the class' methods.

Local variables are created in the state machine's `__init__` method. Because a state machine has to be set up with an EventInbox and other bookkeeping, the `__init__` of the class' superclass should be called first. Local variables can be referred to as `self.variable`. The `__init__` method is also where the initial state should be indicated.

Thesis' syntax:

```
machine example {



    var1 = 3;
    var2 = null;



    init state setup {
        var1 = 6;
        var2 = 2 * var1;

        => halt;
    }
}
```

Simulator's syntax:

```
class Example(StateMachine):
    def __init__(self, ctl, ctx):
        super().__init__(ctl, ctx)

        self.var1 = 3
        self.var2 = None

        self.init_state = self.setup

    def setup(self):
        self.var1 = 6
        self.var2 = 2 * self.var1

        return self.halt
```

Although in Python it is possible to create local variables from anywhere within a class, this should not be done in this simulation. This is because the language simulated does not support creating local variables within states.

Instantiating (and starting) machines is done through a method, just as emitting events and registering reactions to them.

Thesis' syntax:

```
machine_var =
    ctl.start(machine, arguments);


emit ("type");
emit ("type", value);
emit ("type") to machine;
emit ("type") => state;


when "type" => state;
when machine emits "type" => state;


ignore when "type";
ignore when machine emits "type";
```

Simulator's syntax:

```
self.machine_var = \
    self.start_machine(machine_class,
                       arguments)

self.emit('type')
self.emit('type', value=value)
self.emit_to('type', machine)
self.emit('type',
          ack_state=self.method)

self.when('type', self.method)
self.when_machine_emits(
    'type', machine, self.method)

self.ignore_when('type')
self.ignore_when_machine_emits(
    'type', machine)
```

The example code from Section 2.1.5 on page 11 looks as follows in the simulator:

```
class Machine1(StateMachine):
    def __init__(self, ctl, ctx, n):
        super().__init__(ctl, ctx)

        self.n = n
        self.i = 0

        self.init_state = self.state_1

    def state_1(self):
        if self.i < self.n:
            # do something
            self.i += 1
            return self.state_1

        return self.halt
```

3.3.2 Debugging

With a tool for testing programs in a programming language comes the need for a debugging toolset. Debugging can be done manually (e.g. by putting `print` statements in state code), but this does not uncover the driving force of the language: events. The simulator comes with two debugging features: *state machine windows* and *stepped execution*.

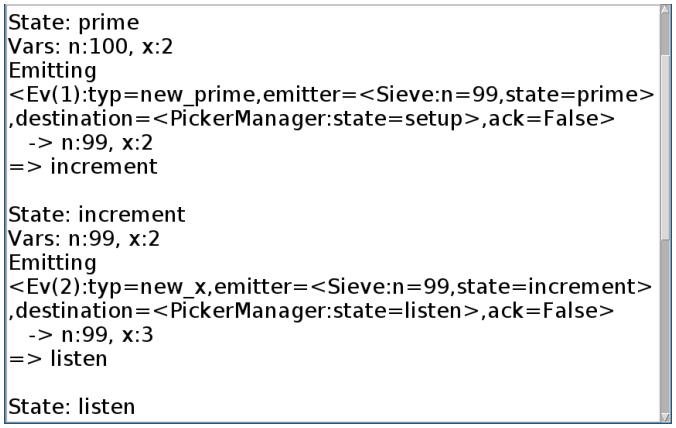

Figure 3.6: Debug window of the *Sieve* machine from the Sieve of Eratosthenes

State machine windows

When a program is run with state machine windows, MC opens a window for each state machine started. This window contains the name of the state machine. Every time a state machine is cycled, new information is displayed in the window. This information comprises the machine's current state and the resulting state of the cycle.

Local variables can be included by adding an *InfoList* to the state machine in the `__init__` method. Every element should be a tuple, containing a Python format string [5] and a string containing the name of the variable to be printed. These entries will be concatenated to a comma-separated string. If any variables have changed after the cycle, their new values will be displayed as well.

```
self.info = [
    ('n:%d', 'n'),
    ('name:%s', 'name')
]
```

When a machine emits an events, this will also be displayed in its window. When a machine is in its listen state, the event it reacts to (if any) will also be displayed. As soon as a machine has executed its halt state and thus is halted, this will be displayed in the window's title. When a program is finished (or stopped by the user) the debug windows remain for inspection. An example window is shown in Fig. 3.6.

Stepped execution

Bugs can occur in many forms, such as typing errors in events and wrong state transitions due to if-statements. Finding these bugs through state machine debug windows can be difficult, especially with complex state machines that cycle through many states.

The simulator supports stepped execution. This means that after each cycle, execution will be stopped until the user presses enter. This way, the user can debug their program step by step, showing every step in debug windows. This makes finding bugs less difficult, especially when the user keeps their state machine diagrams at hand.

3.4 Experiments

3.4.1 Synchronisation problem

As explained in Section 2.3.6 on page 14 the language contains event acknowledgements. One of the very first experiments surfaced the need for this important aspect of the language. The problem, called the *synchronisation problem*, was actually discovered by hand, but was later replicated using the simulator.

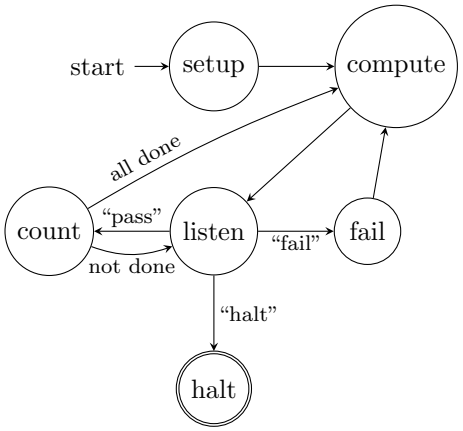

Figure 3.7: Synchronisation machine A. State workings: *setup* starts multiple machines B; *compute* initialises computation, subscribes to reactions and emits "run" to machines B; *count* counts number of pass signals; *fail* disables reactions.

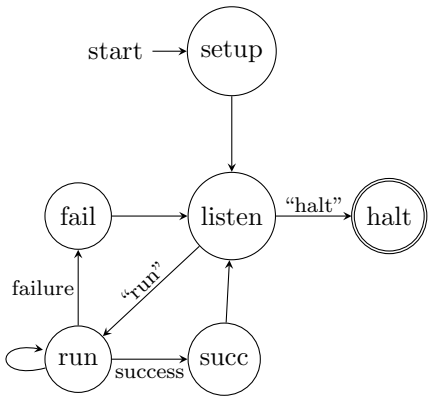

Figure 3.8: Synchronisation machine B. State workings: *setup* subscribes to reactions; *run* loops for a while and decides success or failure; *fail* emits "fail" to context; *succ* emits "pass" to context.

Suppose there are two types of state machines, machine A and machine B. Their behaviours are displayed in Figures 3.7 and 3.8.

Machine A does some computation and manages multiple instances of machine B which do some filtering on results. A runs the B machines simultaneously, which either fail or pass through an event.

Passes are good, so when all B machines have passed A can produce a new computation. Fails are not so good. As soon as A receives a failure, the results from the rest of the B machines are unimportant, so they can be ignored. However, the computations B performs are important, so they should still be run.

Suppose these computations of B take quite a while, except the first failure. At this point A will ignore future events from B machines and initiate a new round of computation. It will register the same reactions as before and tell the B machines to start their computation again.

In the meantime, however, the still running B machines have still emitted their fails and passes. All of these will end up in A's EventInbox. So when A has left the compute state and entered the listen state, it will treat these events as if they originate from the latest round of computation.

This can lead to unexpected behaviour and faulty computations. It is possible that A reacts to a failure, while actually all B machines pass the relevant computation. The opposite is possible as well, if A counts enough passes, while actually a B emitted a failure.

The solution to this problem is *synchronisation*. Before A starts a new computation, it will have to wait for all B machines to finish. This can be done by emitting a "sync" event with an acknowledgement. In stead of immediately turning the reactions back on, this happens per machine upon receiving the acknowledgement. The working machines A' and B' are displayed in Figures 3.9 and 3.10.

3.4.2 Starting problem

As explained in Section 2.3.4 on page 13, filtering of events is done by state machines. In an earlier design this filtering was done by MC upon event distribution. However, testing programs in the simulator showed a problem with this. This does show the importance of the simulator, as the problem would not have been found with only manual testing.

Suppose event filtering is done by MC upon event distribution. Figures 3.11 and 3.12 show two state machines A and B. A starts a machine B and immediately emits "run" to it. B's first

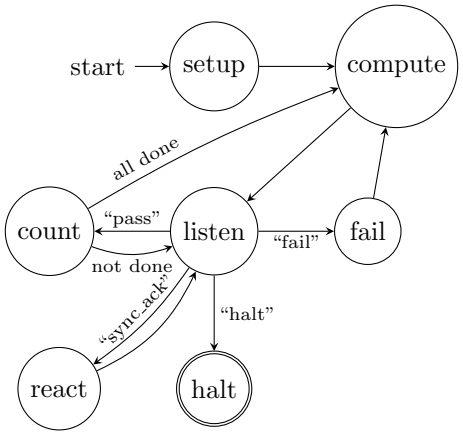

Figure 3.9: Synchronisation machine A'. State workings: *setup* starts multiple machines B; *compute* initialises computation and emits "sync" to machines B; *count* counts number of pass signals; *fail* disables reactions; *react* subscribes reactions to emitter.

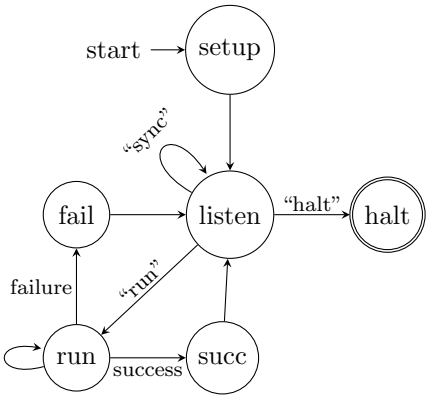

Figure 3.10: Synchronisation machine B'. State workings: *setup* subscribes to reactions, including "sync" to *listen*; *run* loops for a while and decides success or failure; *fail* emits "fail" to context; *succ* emits "pass" to context.

state registers a reaction to this event.

However, MC is implemented in the simulator to distribute all events before each cycle. This means that the "run" event will be distributed *before* B has been able to register its reaction to it. Thus, MC will not put the event in B's EventInbox. This means that B will never perform its computation and neither A or B will ever halt.

There are multiple ways to solve this problem without changing the language design. The implementation could be changed, so that event distribution is smart and will be scheduled to avoid this problem. However, this would make the behaviour of the language depend on the implementation, which is undesirable.

The problem can be also avoided by letting A wait for B to setup its reactions through an event and extra states. This, however, requires extra states just to work around a problem which can occur in many programs.

Thus, by changing the language design to filter events in a machine's listen state, the entire problem has been removed. A downside is that machines will need to access MC's reaction maps frequently, which in a concurrent implementation could cause overhead. It is also possible to put the reaction maps in the state machines themselves, but this would cause each machine to have more overhead for bookkeeping.

3.4.3 Turing completeness

An important aspect of programming language design is Turing completeness. It originates from the Church-Turing Thesis. This thesis states that "a function is effectively computable if its values can be found by some purely mechanical process." Such a purely mechanical process is the Turing machine.

The Turing machine consists of an infinitely large tape (i.e. infinite memory), a tape head, a finite number of states and a state table. The machine is always in a state. An entry in the state table contains an action (a direction in which to move the tape head) and a state transition. The state head reads a character from the tape, which in combination with the current state is used to determine the action and next state. The Turing machine can compute all effectively computable functions.

A programming language is Turing complete if and only if it can be used to compute all effectively computable functions. As the Turing machine is Turing complete, a programming language is Turing complete if it can be simulated in a Turing machine. This also works vice

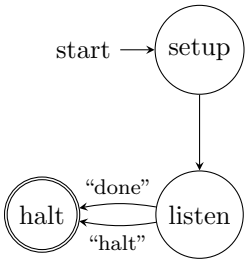

Figure 3.11: Start problem machine A. State workings: *setup* starts a machine B, registers reaction to "done" and emits "run".

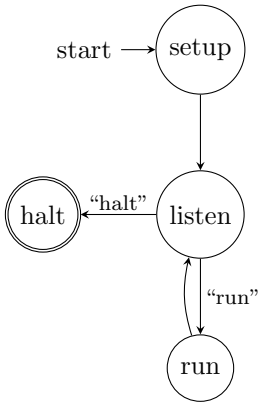

Figure 3.12: Start problem machine B. State workings: *setup* registers reaction to "run"; *run* does some computation and emits "done".

versa, i.e. if a Turing machine can be simulated in the programming language.

The (simulation of the) programming language described in this thesis is Turing complete. A simulation of the Turing machine has successfully been programmed. Although no computer exists with infinite memory, this requirement is often ignored. Another way to look at it is that the language is required to be able to maintain an arbitrary amount of variables. The simulation consists of two state machines: the $TuringMachine$ (see Fig. 3.13) and the $TuringTape$ (see Fig. 3.14).

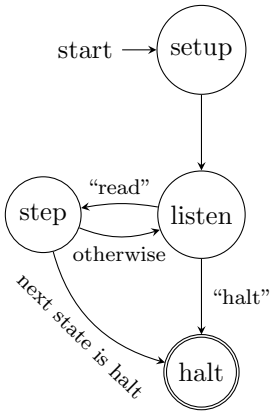

Figure 3.13: State machine *TuringMachine*. State workings: *setup* starts *TuringTape* and registers reaction to "read"; *step* determines action and next state and emits the action.

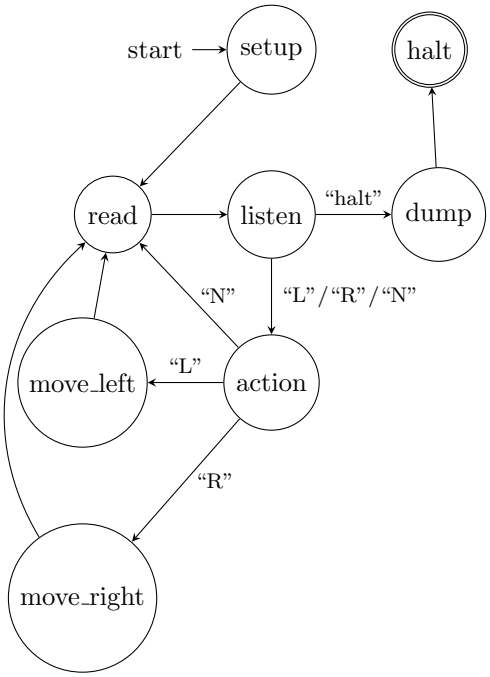

Figure 3.14: State machine *TuringTape*. State workings: *setup* registers reactions to actions; *read* emits the character under the tape head; *action* writes the action's character; *move_left* moves the tape head left, making extra space if necessary; *move_right* moves the tape head right, making extra space if necessary; *dump* prints the full contents of the tape.

CHAPTER 4

Discussion

The simulation shows that the language design is suitable for purely event-driven programming. Problems found through experimentation could be solved by changing the language design.

The language design successfully avoids inversion of control. A programmer is fully in control of The control flow of a program. However, designing programs is still difficult. The language requires a different mindset from regular sequential programming. Drawing state machines and turning them into code is a good way to design programs, although complex machines require complex drawings.

The usage of state machines as independent units supports the notion that event-driven programming is suitable for concurrent execution. This independence also makes thinking in concurrency less complicated. Although the program might stumble upon synchronisation problems, most concurrency is implicit because of the communication through events.

There is, however, a flaw in the language design. When a state machine is in its listen state and no events are emitted, it will still execute code. MC is responsible for the distribution of all events and for managing all state machines. Therefore, MC should be able to determine when a machine is actually unnecessarily executing code. This *busy loop* can be avoided by making machines enter an *idle* state when they are in the listen state and there are no events. MC can then skip cycling machines in the idle state until it has distributed an event.

CHAPTER 5

Related work

In [2], abstractions to the programming language C to write event-driven programs are introduced. Its domain is embedded systems with limited resources. This means that programs should have little overhead and small size. Therefore, the use of explicit state machines is avoided. This is done by introducing *protothreads*. They run inside a function and have minimal overhead. Protothreads can be used to simplify event-driven programs with blocking wait abstractions.

In [6], a library, *libasync*, is introduced that uses event callbacks for blocking I/O. The paper states that thread programming is difficult, thus being more prone to bugs. Although the library is designed to avoid the need for locks and the like, the use of complicated callbacks does not solve the difficulty problem. It does allow the programmer to focus more on their program than on managing threads.

In [1], an event-driven extension to Scala is proposed. An *actor* library is available for Scale, which is an abstraction to concurrency through threads. The paper states problems with event-driven programming based on callbacks and with programming with threads. It changes the actor library to an event-driven library.

The research in [7] is similar to the research in this thesis. The paper describes a programming language *P*, comprised of state machines that communicate through events. An important difference is that in P events are handled statically in states. This means that every state contains state transitions for events, opposed to dynamically assigning state transitions through the listen state. P also supports program validation in its compiler.

CHAPTER 6

Further research

The programming language design described in this thesis is merely the foundation of an actual programming language. There are two options: extend the design so that a compiler can be made for it, or embed the language in another language (such as Haskell).

Depending on the language of choice, a syntax has to be defined. Also, choices need to be made on how events are to be represented. To make a compiler, there are more decisions to be made. Will the language be statically or dynamically typed? How will state machines be represented in memory and how can they be accessed from MC?

In both cases, the suitability for concurrency needs to be kept in mind. This means that the way values are passed with events should be devised. In the Python simulator it is possible to pass any value, even though that value is a memory reference. When state machines are implemented as independent thread-like units, simply passing references should not be possible in such a naive manner. However, leaving the programmer responsible for safe concurrency should be avoided. Designing a system for this is a research on its own.

Another field of further research is the verification of purely event-driven programs. Just as the validation in P [7], it should be possible to validate programs in the language described in this thesis. A possibility for this is to provide a projection onto Program Algebra (PGA), which already has support for state machines [8]. In [9, p. 30-64], this is called *services*. In [10], a good example of the simulation of Turing machines is provided. A program's semantics can then be verified through Basic Polarized Process Algebra (BPPA). This can also be done by simulating purely event-driven programming in Process Specification Formalism (PSF) [11].

CHAPTER 7

Conclusion

The research question this thesis has tried to answer is: is it possible to program using only events? And if so, what would it look like? Can inversion of control be avoided? Can concurrency be supported implicitly?

This thesis successfully describes a purely-event driven programming language. Although there is much to be done to be able to actually program in the language, concurrency is implicit and clear. Theoretical overhead is kept to a minimum and inversion of control is avoided.

This novel type of programming gives way to new algorithms and redesigns of existing algorithms. Experimentation has shown that this type of purely event-driven programming is very suitable for algorithms that have an obvious master/worker structure (such as the Sieve of Eratosthenes). The language is also suitable to program systems that are event-driven by nature (for example an elevator or user interface). Although the language can be used to program algorithms that require some form of sequentiality, this seems more of a hassle than to have immediate benefits. Nevertheless, the language can be used as a general purpose programming language.

Bibliography

[1] Philipp Haller and Martin Odersky. 'Event-based programming without inversion of control'. In: *Modular Programming Languages*. Springer, 2006, pp. 4–22.

[2] Adam Dunkels et al. 'Protothreads: simplifying event-driven programming of memory-constrained embedded systems'. In: *Proceedings of the 4th international conference on Embedded networked sensor systems*. Acm. 2006, pp. 29–42.

[3] Gregor Hohpe. 'Programming without a call stack-event-driven architectures'. In: *Objekt Spektrum* (2006).

[4] SKopp. *Sieve of Eratosthenes animation*. Creative Commons Attribution-Share Alike 3.0 Unported license. 2011. URL: https://commons.wikimedia.org/wiki/File:Sieve_of_Eratosthenes_animation.gif (visited on 06/06/2016).

[5] Python Software Foundation. *Python 3.4.4 documentation - Built-in types: printf-style String Formatting*. 2016. URL: https://docs.python.org/3.4/library/stdtypes.html#printf-style-string-formatting (visited on 06/06/2016).

[6] Frank Dabek et al. 'Event-driven programming for robust software'. In: *Proceedings of the 10th workshop on ACM SIGOPS European workshop*. ACM. 2002, pp. 186–189.

[7] Ankush Desai et al. 'P: safe asynchronous event-driven programming'. In: *ACM SIGPLAN Notices* 48.6 (2013), pp. 321–332.

[8] Jan A Bergstra and Alban Ponse. 'Combining programs and state machines'. In: *The Journal of Logic and Algebraic Programming* 51.2 (2002), pp. 175–192.

[9] Jan A Bergstra and Cornelis A Middelburg. *Instruction Sequences for Computer Science*. Vol. 2. Springer Science & Business Media, 2012.

[10] Jan A Bergstra and Alban Ponse. 'Execution architectures for program algebra'. In: *Journal of Applied Logic* 5.1 (2007), pp. 170–192.

[11] Bob Diertens. 'Software Engineering with Process Algebra'. PhD thesis. University of Amsterdam, 2009.

Acronyms

BPPA Basic Polarized Process Algebra. 31

MC Machine Control. 9–15, 19, 20, 22–24, 27, 31

PGA Program Algebra. 31

PSF Process Specification Formalism. 31

UI User Interface. 7

Appendices

APPENDIX A

Sieve of Eratosthenes simulator code

```
from simulator import MachineControl, StateMachine


class Sieve(StateMachine):
    def __init__(self, ctl, ctx, n):
        super().__init__(ctl, ctx)

        self.n = n

        self.x = 2
        self.manager = None

        self.info = [
            ('n:%d', 'n'),
            ('x:%d', 'x'),
        ]

        self.init_state = self.setup

    def __repr__(self):
        return '<Sieve:n=%d,state=%s>' % (self.n, self.current_state.__name__)

    def setup(self):
        if self.n < 0:
            return self.halt

        self.manager = self.start_machine(PickerManager)

        self.when_machine_emits('pass', self.manager, self.prime)
        self.when_machine_emits('fail', self.manager, self.increment)

        return self.prime

    def prime(self):
        self.n -= 1

        print('FOUND PRIME %d, %d left' % (self.x, self.n))

        if self.n == 0:
```

```
            return self.halt

        self.emit_to(self.manager, 'new_prime', value=self.x)

        return self.increment

    def increment(self):
        self.x += 1

        self.emit_to(self.manager, 'new_x', value=self.x)


class PickerManager(StateMachine):
    def __init__(self, ctl, ctx):
        super().__init__(ctl, ctx)

        self.pickers = []
        self.i = 0
        self.togo = 0
        self.current_x = 0

        self.init_state = self.setup

    def __repr__(self):
        return '<PickerManager:state=%s>' % (self.current_state.__name__)

    def setup(self):
        self.when_machine_emits('new_prime', self.ctx, self.new_prime)
        self.when_machine_emits('new_x', self.ctx, self.new_x)

    def new_prime(self):
        self.pickers.append(self.start_machine(Picker, self.event.value))

    def new_x(self):
        self.i = 0
        self.togo = len(self.pickers)
        self.current_x = self.event.value

        return self.run_pickers

    def run_pickers(self):
        if self.i < len(self.pickers):
            self.emit_to(self.pickers[self.i], 'run', value=self.current_x,
                         ack_state=self.listen_picker)
            self.i += 1

            return self.run_pickers

    def listen_picker(self):
        if self.event.value == self.current_x:
            m = self.event.emitter
            self.when_machine_emits('pass', m, self.got_pass)
            self.when_machine_emits('fail', m, self.got_fail)

    def got_pass(self):
```

```
        self.togo -= 1
        if self.togo == 0:
            self.emit_to(self.ctx, 'pass')

    def got_fail(self):
        self.emit_to(self.ctx, 'fail')

        self.i = 0
        return self.unlisten_pickers

    def unlisten_pickers(self):
        if self.i < len(self.pickers):
            m = self.pickers[self.i]
            self.ignore_when_machine_emits('pass', m)
            self.ignore_when_machine_emits('fail', m)

            self.i += 1
            return self.unlisten_pickers


class Picker(StateMachine):
    def __init__(self, ctl, ctx, x):
        super().__init__(ctl, ctx)

        self.x = x
        self.count = 0

        self.info = [
            ('x:%d', 'x'),
            ('count:%d', 'count'),
        ]

        self.init_state = self.setup

    def __repr__(self):
        return '<Picker:x=%d,count=%d,state=%s>' % (
            self.x, self.count, self.current_state.__name__)

    def setup(self):
        self.when('run', self.run)

    def run(self):
        self.count += 1

        if self.count == self.x:
            self.count = 0
            self.emit('fail')

        else:
            self.emit('pass')


if __name__ == '__main__':
    ctl = MachineControl(debug=False, step=False)
    ctl.run(Sieve, 100)
```

APPENDIX B

Python simulation documentation

Simulates a purely event-driven programming language.

This simulator is part of a computer science bachelor's thesis for the University of Amsterdam, by Bas van den Heuvel. It follows all concepts introduced in this thesis. The purpose is to be able to test and refine those concepts.

The scheduling method is deterministically sequential. No concurrent execution is implemented.

Classes:

- MachineControl: manages and schedules state machines and events
- Event: event for communication between state machines
- StateMachine: superclass for all possible state machines

class simulator.simulator.**Event**(*typ*, *emitter*, *value=None*, *destination=None*, *ack=False*)
An event for interaction between state machines.

__init__(*typ*, *emitter*, *value=None*, *destination=None*, *ack=False*)
Initialize the event.

Parameters

- **typ** – the event's type string
- **emitter** – the StateMachine emitting the event

Keyword Arguments

- **value** – value to transmit (default None)
- **destination** – the StateMachine the event should end up with (default None)
- **ack** – whether the receiving machine should emit an acknowledgement (default False)

class simulator.simulator.**MachineControl**(*debug=False*, *step=False*)
Manage and schedule state machines and events.

Every program created with this simulator should have an instance of this class. Starting a program goes through this instance, as well as instantiating state machines and sending events.

Execution of its machine's states happens through cycles. The scheduling for this is sequential and very minimalistic: the first machine in its queue gets cycled after which it gets replaced at the end of the queue.

It can be said that there is no event scheduling. Before each state cycle, all events in the event buss are distributed to their respective state machines.

__init__(*debug=False*, *step=False*)
Initialize a machine control.

It setups up a machine list, which in this implementation is a queue. Reaction maps are created as dictionaries, and the event buss is another queue.

The *ctx* variable is not yet set. This happens when the simulator is started.

Keyword Arguments

- **debug** – opens a window for each state machine showing state and event information if True (default True)
- **step** – allows one to cycle stepwise (default False)

add_event_reaction(*typ*, *reactor*, *state*)
Add a reaction to an event.

Parameters

- **typ** – the event's type string

- **reactor** – the StateMachine that should react
- **state** – the state the machine should transition to, a method

add_machine_reaction(*typ*, *emitter*, *reactor*, *state*)
Add a reaction to a state machine's event.

Parameters

- **typ** – the event's type string
- **emitter** – the event's emitting StateMachine
- **reactor** – the StateMachine that should react
- **state** – the state the machine should transition to, a method

cycle()
Distribute events, cycle a machine and return whether any are left.

When the machine queue is empty, False is returned. Otherwise, True is returned.

If debuggin is on, the cycles machine's state before and after the cycle is shown in the machine's debuggin window, accompanied by any variables indicated in the machine. If these variables have changed after the cycle, the changed values are shown as well.

debug_aftercycle(*machine*, *p_state*, *n_state*, *p_var_str*)
Send info to a machine's debug window after a cycle.

First the machine's variables are compared to its variables before the cycle. If they are changed, they are displayed. If the machine was in its listen state and reacted to an event, this event is displayed. Finally, if the cycle resulted in a state transition, the new state is displayed.

If the machine cycled its halt state, nothing is to be done.

Parameters

- **machine** – the StateMachine to show debug information for
- **p_state** – the machine's state before the cycle
- **n_state** – the machine's state after the cycle
- **p_var_str** – the machine's variable string before the cycle

debug_precycle(*machine*)
Send info to a machine's debug window before a cycle.

First the machine's current state is show. After this, if the machine has indicitated any variables as information, these are shown as well.

Parameters **machine** – the StateMachine to show debug information for

distribute_events()
Distribute an event to machines and return whether any are left.

If an event has a destination and that destination is still alive (i.e. not halted), the event is put into that machine's inbox. Otherwise, the event is put into the inbox of all live machines, except the event's emitter.

If an event has been distributed, True is returned. Otherwise, False is returned.

emit(*event*)
Add an event to the event buss.

If debugging is on, the event is displayed in the emitter's debug window.

Parameters **event** – the to be emitted Event

filter_event (*machine*, *event*)
Returns a state if a machine should react to an event.

If a reaction exists, the machine's reaction state is returned. Otherwise, None is returned.

First machine reactions is checked, because such reactions are more specific and thus have priority.

Parameters

- **machine** – the reacting state machine, a StateMachine
- **event** – the event to be checked, an Event

halt (*machine*)
Halt a machine.

If debuggin is on, the machine's debuggin window's title is altered to include 'HALTED' and the window's stdin pipe is closed.

remove_event_reaction (*typ*, *reactor*)
Remove a reaction to an event.

Parameters

- **typ** – the event's type string
- **reactor** – the StateMachine that should ignore the event

remove_machine_reaction (*typ*, *emitter*, *reactor*)
Remove a reaction to a state machine's event.

Parameters

- **typ** – the event's type string
- **emitter** – the event's emitting StateMachine
- **reactor** – the StateMachine that should ignore the event

reset ()
Reset machine control.

This prepares it for a next run. Python garbage collects itself, but in an actual implementation, all these fields need to be emptied carefully.

run (*machine_cls*, **args*, ***kwargs*)
Start a state machine and cycle until all machines have halted.

A context is created for this first machine, by instantiating the StateMachine superclass without a context.

Parameters

- **machine_cls** – a StateMachine subclass
- ***args/**kwargs** – any arguments the state machine takes

start_machine (*machine_cls*, *ctx*, **args*, ***kwargs*)
Start a state machine.

Initializes a machine, given arbitrary arguments, and adds it to the machine queue. After this, event reaction to 'halt' is added.

If debugging is turned on, this also starts a debug window, able to show state and event information.

Parameters

- **machine_cls** – a StateMachine subclass

- **ctx** – the state machine that starts this new machine
- ***args/**kwargs** – any arguments the state machine takes

class simulator.simulator.**StateMachine**(*ctl*, *ctx*)

Represent a state machine.

To create purely event-driven programs, one can subclass this class. The *__init__* method should be extended with local variables and an initial state, but first call *super().__init__()* to prepare the machine.

States can be implemented by adding methods to the class. The initial state can be indicated by setting *self.init_state* to the preferred state method in *__init__*. Loops are not impossible, but should not be used as they do not exist in the language proposed by this thesis.

Do not override *listen* and *halt*, this will break the simulator. However, referring to both states is no problem (and usually necessary).

The current event can be referred to through *self.event*. Do not mutate this variable.

__init__(*ctl*, *ctx*)

Initialize the state machine.

Prepares the machine for execution and prepares event processing necessities.

Parameters

- **ctl** – a MachineControl instance
- **ctx** – the machine's context, a StateMachine

cycle()

Run the current state and determine the next.

If no next state is obtained, the new state will be 'listen'.

emit(*typ*, *value=None*)

Emit an event.

Parameters **typ** – the event's type string

Keyword Arguments **value** – value to transmit with the event (default None)

emit_to(*destination*, *typ*, *value=None*, *ack_state=None*)

Emit an event to a machine.

Parameters

- **destination** – the StateMachine to send the event to
- **typ** – the event's type string

Keyword Arguments

- **value** – value to transmit with the event (default None)
- **ack_state** – a state for acknowledgement, a method

If *ack_state* is given, the receiving machine will send an acknowledgement event. When the emitting machine recieves this event, it will transition to the given state.

filter_event(*event*)

Return a state if a reaction to the event exists.

Parameters **event** – the Event

halt()
Halt state for all machines.

Do not extend or override this method.

First emits 'halt', which halts all child machines. Then MachineControl is told to halt the machine.

ignore_when(*typ*)
Remove an event reaction.

Besides ignoring further such events, all events from the given machine and of the given type in the machine's inbox are removed.

Parameters **type** – the event's type string

ignore_when_machine_emits(*typ*, *machine*)
Remove a machine event reaction.

Besides ignoring further such events, all events from the given machine and of the given type in the machine's inbox are removed.

Parameters

- **typ** – the event's type string
- **machine** – the event's emitting StateMachine

listen()
Listen state for all machines.

Do not extend or override this method.

Checks the event inbox for any events and possible reactions. If an acknowledgement is required, this is sent.

start_machine(*machine_cls*, **args*, ***kwargs*)
Instantiate and start a machine.

Parameters

- **machine_cls** – a StateMachine subclass
- ***args/**kwargs** – any arguments the state machine takes

var_str()
Create a string of formatted variables.

self.info should contain a list of tuples with a format string and the name of a variable. This method aggregates them into a comma-separated string containing these formatted values.

when(*typ*, *state*)
Add an event reaction.

Parameters

- **typ** – the event's type string
- **state** – the state to transition to, a method

when_machine_emits(*typ*, *machine*, *state*)
Add a machine event reaction.

Parameters

- **typ** – the event's type string
- **machine** – the emitting StateMachine

- **state** – the state to transition to, a method

class simulator.debug_window.**DebugWindow**(*title='State Machine'*)
Create a Tk window for displaying debug messages.

The *Window* class in this script is the actual window. By invoking this file directly, such a window is created. This window reads from stdin. This class does exactly that. It runs this script as a subprocess, linking its stdin to a writable buffer.

__init__(*title='State Machine'*)
Initialize a debug window.

Opens a Tk window in a subprocess, in text-mode which allows the stdin pipe to be used for text directly.

Keyword Arguments **title** – the window's initial title (default 'State Machine')

close()
Close the window's stdin pipe.

This does not actually close the window, only the stream. The window is kept open so the user can analyse states even after a program is finished.

set_title(*title*)
Set the window's title.

Parameters **title** – the title

write(*text*)
Write a line to the window.

The window might have been closed by the user or some different event. This is ignored.

Parameters **text** – text excluding newline

class simulator.debug_window.**Window**
Show a Tk window with scrollable text from stdin.

Checks stdin for a new line every one millisecond. If the line starts with a '#', the rest of the line is used as a new title for the window. Otherwise, the line is appended to the textfield, including the newline character.

__init__()
Initialize the window.

Creates a frame, holding a srollable textfield. Finally reading from stdin is initiated.

do_read()
Try to read a line from stdin.

process_line(*line*)
Process a line for debug display.

If a line starts with '#', change the window's title. Otherwise, write the line to the textbox.

Parameters **line** – the line to be processed, including newline character

write_text(*text*)
Write text to the end of the textfield.

Parameters **text** – the text to be added to the textfield.

simulator.debug_window.**main**()
Make stdin nonblocking and open a window.

simulator.debug_window.**make_nonblocking**(*fh*)
Make a file nonblocking.

fcntl is a C system call, used to modify file descriptors. The operation used (F_SETFL) sets the file descriptor's flags.

The argument to this function call uses F_GETFL, which gets the currently set flags. These are combined with a new flag: O_NONBLOCK. This flag makes sure no calls to the file cause the process to wait, i.e. nonblocking.