Poker: Visual Instrumentation of Reactive Programs With Programmable Probes

Cloé Descheemaeker
cloe.descheemaeker@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

Sam Van den Vonder
sam.van.den.vonder@vub.be
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium

Thierry Renaux
thierry.renaux@vub.be
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium

Wolfgang De Meuter
wolfgang.de.meuter@vub.be
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium

Abstract

This paper presents Poker, a visual instrumentation platform for reactive programs. Similar to other platforms, Poker features a visual dashboard that allows the programmer to inspect the flow of values through the reactive program. The novelty of Poker is that: (a) It features a canvas of so-called *probes* that can be dynamically wired into a running reactive program in order to instrument the running system. (b) In addition to focusing on the values flowing through the program, a probe can measure a particular property about the way these values flow through the instrumented program. (c) The set of probes is open because a probe is programmed in the same language as the instrumented program.

Poker is implemented for Stella [7], an experimental reactive programming language. The paper uses an application written in Stella to motivate the concepts provided by Poker. We show 4 different probes that help us understand the behaviour of the application and we measure the overhead of using Poker on the running application with some preliminary benchmarks.

Keywords: Reactive Programming, Debugging, Instrumentation, Visual Programming, Actors, Reactors

ACM Reference Format:
Cloé Descheemaeker, Sam Van den Vonder, Thierry Renaux, and Wolfgang De Meuter. 2021. Poker: Visual Instrumentation of Reactive Programs With Programmable Probes. In *Proceedings of REBLS '21: Workshop on Reactive and Event-Based Languages and Systems (REBLS '21).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
REBLS '21, October 17–22, 2021, Chicago, IL
© 2021 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Reactive code makes up a significant portion of today's software applications. Examples can be found in web-based GUIs, Cyber Physical Systems, robots and web-based collaborative platforms. Reactive code is hard to understand and debug because its behaviour is typically driven by incoming third-party data: It is typically implemented in a declarative reactive programming language which makes "operational understanding" harder and because it is often impossible to switch off; i.e. it is perpetually running. But even if we do allow a reactive application to be halted for inspection, it is typically hard to understand the behaviour of the entire reactive application by merely inspecting "the current state" of every single reactive operator that is part of the application (or nodes if we think about the DAG that corresponds to the application) at a particular moment in time. After all, the state of every single node only reveals a very small portion of the state of the entire system. This is the reason why researchers have developed visual tools that allow a programmer to inspect the state changes of said operators by visualising the flow of values. E.g., [3] allows the developer to visualise a running reactive program in terms of so-called marble diagrams. A number of variations on this theme can be found and we present an overview in Section 2.

In software engineering a distinction is made between the functional and the non-functional aspects of a system [9]. The former generally constitute *what* the system is doing while the latter constitute *the way* it is performing that task. Examples of non-functional requirements are memory consumption, throughput, speed, security, authentication, etc. With this terminology, the aforementioned visualisation tools only allows a developer to visualise and understand the functional aspects of a reactive program. In this paper we

motivate the need for more powerful instrumentation tools that allow a developer to also understand the non-functional aspects of their program.

In this paper, we present Poker[1][2]. Poker is an instrumentation platform for instrumenting the functional as well as the non-functional aspects of reactive programs. Poker has the following properties:

Visual Probes Poker features a number of visual components called *probes*. Poker visualises a running reactive program by displaying a DAG-representation of said program. Poker's probes can be wired into and out of the running DAG, and every probe displays a functional or a non-functional property of the running reactive program. Hence, probes are like measuring instruments or gauges that a programmer can add and remove from a running program in a visual way. In Section 4 we will demonstrate probes for visualising the values flowing through a program as well as probes for visualising the performance and the throughput of the system.

Open Set of Probes The set of Poker's probes is not limited to the built-in ones. Because probes are written in the same language as the reactive program under inspection, an expert developer can easily extend their suite of probes. At startup time, Poker loads its set of probes from a file. Hence, by simply extending this file with code implementing additional probes, Poker is automatically extended. In section Section 5, we show how to extend Poker with an intelligent probe that not only instruments the running application but which also modifies the behaviour of the program. This allows us to experiment with alternative implementations while understanding and inspecting the behaviour of a running reactive application.

A central question that needs to be understood if we instrument a reactive system is to what degree the instrumentation code itself will affect the instrumented program. In physics this disturbance of an observed system by the act of observation is known as **The Observer Effect** [6]. We will consider this both from a qualitative as well as from a quantitative point of view. The former is about how the presence or absence of Poker probes affects the semantics of the running system. The latter is about how the presence of absence of Poker probes affects the run-time performance of the system.

The paper is structured as follows. In Section 2 we motivate the need for richer tool sets to understand the behaviour of reactive systems. In Section 3 we briefly introduce Stella, the programming language that was used as the laboratory for the research. Stella features both actors and reactors and having access to its full tool chain allowed us to do the research. In Section 4 we introduce Poker. We show the visual instrumentation platform with the built-in set of instruments. In Section 5 we show how to extend the current instrument suite. In Section 6 we discuss how Poker and Stella interact, and in Section 7 we elaborate on the observer effect induced by some Poker probes. Sections 8 and 9 discuss Poker's limitations and related work respectively, and finally Section 10 concludes the paper.

2 Problem Statement

Programmers require the means for gaining "operational understanding" of reactive systems, both of their functional aspects and their non-functional aspects. A survey among programmers experienced in developing reactive programs identified 4 overarching practices developers perform when debugging reactive programs [3]. Those related to acquiring an operational understanding are "*Gaining high-level overview of the reactive structure*" and "*Understanding dependencies [between time-varying values]*".

Both of the aforementioned practices are supported by existing visualisation tools for debugging reactive systems. Those debugging tools excel at granting programmers an understanding of reactive programs at the **micro-level**. Often this is achieved by transforming source code into a visual representation like a DAG, e.g., the stop-the-world debugger for reactive programs proposed in [21], or "*marble diagrams*" proposed in [3]. Such tools then visualise the way in which data elements flow through the running program. In general we say that they offer a very fine-grained view of the functional requirements of a reactive program, i.e., on the level of individual program statements (e.g., function applications). Furthermore we say that they are a "hands-off" approach, since they do not require the developer to modify the code of the application under inspection. This hands-off nature of the tools is important, since the extra effort required by "hand-on" tools holds back engineers from applying the tools [2].

In industry, complex (distributed) systems are typically monitored using "dashboards" that display metrics such as average response times, network usage, and disk usage of their servers, services, etc. We call these **macro-level** metrics since they are more coarse-grained. These dashboards are commonly available for large frameworks used for building and orchestrating distributed services, e.g., Akka [19] and Kafka [13], and often take the form of commercial products[3] which visualise the macro-level metrics of an entire distributed application as a number of data points and time series on a website. The metrics gathered on those dashboards are not targeted at a deep, micro-level inspection of specific parts of the reactive application in response to a specific desire for insight. Instead, they are "always on", supported by a service

[1] *Poker* (noun): one that pokes.

[2] Since Poker is a visual tool, we have made an accompanying video in order to illustrate its working. https://youtu.be/5cqDsq4LikA. Video also available at https://doi.org/10.5281/zenodo.5196223.

[3] Examples include Grafana (https://grafana.com/), DynaTrace (https://www.dynatrace.com/monitoring/technologies/java-monitoring/akka/), Kamon (https://kamon.io/solutions/monitoring-for-akka/), and Phobos (https://phobos.petabridge.com/)

running alongside the application from the start. These types of frameworks are "hands-on": Data is exposed to the dashboard by explicit library calls added to the application source code. Hence, these types of dashboards are the inverse of micro-level reactive debugging tools: After provisioning a reactive program and the infrastructure that it is running on, they typically offer a generic and coarse-grained view of the non-functional requirements of a the reactive system.

Currently, hands-off tools such as those proposed in [21] and [3] give a developer insight into the inner workings of a system's *functional* requirements. In other words, they excel at showing their users *what* a system is currently doing (e.g., "Are the computed values correct?"). For programmers to gain an operational understanding of their system, we argue that it is equally valuable to gain insight into the *non-functional* requirements, as demonstrated by the large industrial dashboards. In other words, developers should also gain insights into *the way* the system is behaving, but *without* having to modify the code of the instrumented application. The existing work on a reactive debugger [21] partly identified this necessity by visualising metrics about the "performance" of a particular node in the DAG, but this metric is built-in and not extensible.

The vision behind Poker is to enrich the existing toolbox of reactive programmers by allowing them to inspect the data flow of their reactive program to collect both functional and non-functional metrics such as throughput and performance. Furthermore, since the goal is to help them gain a better operational understanding of *their* specific programs, we propose an *open* instrumentation platform that is easily extendable via new metrics.

3 The Actor-Reactor Model

Poker was implemented in TypeScript as part of the tool suite for the experimental programming language Stella [7] (also written in TypeScript) that is used by the authors to research new mechanisms for reactive programming. The overall vision behind Poker is the same as the one behind Smalltalk [10] and Self [23], two programming languages that are intricately connected with their IDE. As we will demonstrate in Section 5, Poker can be extended in Stella as well. This section gives a brief overview of Stella.

There is a strong correspondence between reactive programs and their representation as a data flow DAG. Besides being used in the implementation of the interpreter or compiler, reactive code is often easily visualised as a DAG, which may aid its understanding [3, 21]. Poker also relies on this correspondence to visualise a reactive program, but not at the usual granularity of individual program statements (e.g., function applications). Instead we assume the *Actor-Reactor Model*, where programs consist of *actors* and *reactors* that interact via data streams. The Actor-Reactor Model was first implemented in Stella, and Poker is designed to instrument

```
1 (def a 1) // (def <identifier> <expression>)
2 (set! a 2) // (set! <identifier> <expression>)
3 // (if <condition> <consequent> <alternative>)
4 (if (eq? a 1) (println! "y") (println! "n"))
```

Listing 1. Examples of basic Stella expressions.

```
1 (def-actor Main
2   (def-constructor (start env)
3     (println! "Hello World!")))
```

Listing 2. A "Hello World!" program in Stella.

Stella programs on the granularity of its so-called actors and reactors.

3.1 Base Language

Stella has two layers: a sequential object-oriented (OO) base language, and the concurrent level of actors and reactors. The sequential base language contains objects such as numbers, strings (e.g., `"hello"`) and symbols (e.g., `'hello`), as well as method invocations on objects and a number of special forms (e.g., to spawn actors and reactors). Examples of a variable definition via `def`, assignment via `set!` and a conditional using `if` are given in Listing 1. Method invocations follow operator prefix notation. E.g., the expression `(eq? a 1)` invokes the method `eq?` on the value of variable `a` with one argument, namely the number `1`. In this case `eq?` checks for object reference equality, and is implemented by the root class `Object`. On the concurrent level, as we will show, actors are responsible for all imperative code, and reactors are responsible for all (purely functional) reactive code. E.g., `set!` cannot be used in code used by a reactor. Their separation is desirable, since side-effects can cause tricky bugs in the reactive program, and have a detrimental effect on behavioural composition [7, 8].

3.2 Actors and "Hello World!"

A Stella program consists of top-level definitions of OO classes (which we do not discuss further), *actor behaviours* ("the class of an actor"), and *reactor behaviours* ("the class of a reactor", i.e., a reactive program). To start a program, the Stella developer implements an actor behaviour called `Main` with a constructor called `start`, e.g., the "Hello World!" program in Listing 2. Its formal parameter `env` may contain values passed from JavaScript (where Stella is started).

Every actor is a concurrent process that has a single behaviour (such as `Main`) and a *mailbox*, i.e., a first-in first-out queue that contains messages. Initially, the Stella interpreter spawns an actor from the `Main` actor behaviour and invokes its `start` constructor. In Listing 2 the constructor of the `Main` actor simply prints "`Hello World!`" to the console.

An atypical feature of Stella's actors is that they can export streams. For example, Listing 3 implements an actor behaviour called `Counter` that implements a stream of monotonically increasing numbers. Line 2 declares a stream called

```
1 (def-actor Counter
2   (def-stream value)
3   (def-fields curr)
4   (def-constructor (init) (set! curr 0))
5   (def-method (increment)
6     (set! curr (+ curr 1))
7     (emit! value curr)))
```

Listing 3. A `Counter` actor behaviour.

```
1 (def-reactor (Add x y)
2   (def res (+ x y))
3   (out res))
```

Listing 4. An `Add` reactor behaviour.

`value` and Line 3 a local field called `curr`. The `init` constructor on Line 4 initializes the `curr` field via an assignment. Line 5 declares a method called `increment` with no arguments. Whenever a `Counter` actor receives an `increment` message, the corresponding method is invoked, and the actor will increment `curr` and `emit!` the new value on its `value` stream. Emitting a new value amounts to sending an asynchronous message to all other actors (and reactors) that are subscribed to the stream.

Actors are spawned by other actors via a `spawn-actor!` statement in the base language, which returns an object of type `ActorReference`. For example, the following snippet spawns an instance of `Counter` with its `init` constructor, and immediately sends it an asynchronous `increment` message.

```
(def ctr (spawn-actor! Counter 'init))
(send! ctr 'increment)
```

3.3 Reactors

Similar to actors, a *reactor* is a process with a mailbox and a *reactor behaviour*, which encapsulates a push-based (functional) reactive program that is compiled to a DAG. A reactor continuously dequeues messages from its mailbox and propagates them through its DAG via an algorithm similar to FrTime [5]. One of the simplest examples is the `Add` reactor behaviour given in Listing 4. A list of sources (inputs) are given at the top, in this case two inputs called `x` and `y`. They can be considered as signals or behaviours (i.e., time-varying values) in other functional reactive programming languages such as FrTime [5], Flapjax [16] and REScala [20]. The body consists of local variable definitions (such as `res`) and method invocations on regular objects (e.g., `+`). Method invocations are recomputed using "or"-semantics, meaning they are reapplied whenever the value of *any* of the given inputs change (using the latest value for the others), e.g., `+` is recomputed whenever `x` *or* `y` changes. The list of output signals is given by `out`, in this case `res`. Whenever the value of `res` changes, it is emitted on a stream called `out` that is exported by the reactor.

```
1 (def-actor Main
2   (def-constructor (start env)
3     (def counter (spawn-actor! Counter 'init))
4     (def adder   (spawn-reactor! Add))
5     (react-to! adder 10 counter.value)
6     (monitor! adder.out 'print!))
7   (def-method (print! val)
8     (println! "output: " val)))
```

Listing 5. Chaining actors and reactors via their streams.

Since reactors are purely functional, only actors can spawn them via `spawn-reactor!` in the base language, which returns a `ReactorReference` object.

```
(def adder (spawn-reactor! Add))
```

3.4 Linking Actors and Reactors via Streams

Actors and reactors are linked via the streams which they export. An example of a complete program that chains the `Counter` and `Add` behaviours is shown in Listing 5, which links the counter's stream to a reactor, and prints the values of the reactor's stream to the console. Here, the `Main` actor first spawns the corresponding `Counter` actor and `Add` reactor (Lines 3 and 4 respectively).

The sources of reactors are changed via a `react-to!` statement on Line 5, which sends an asynchronous message to the reactor that instructs them to change their sources to the number 10 and `counter.value` respectively. Here, the `counter.value` expression yields a reference to the stream called `value` exported by the `counter` actor (returning an object of type `Stream`). Note that this dot-notation can only be used to refer to streams. The reactor will automatically create a dependency on this stream. Thus, whenever the `counter` actor emits a new value to the stream, this value will arrive as an asynchronous message in the mailbox of the reactor. When the reactor processes this message, it changes the value of its corresponding source node (in this case `y`), and propagates the new value through the reactive program in typical reactive programming fashion.

Supplying an actor's stream to a reactor links the actor with the reactor. In the reverse direction, actors can monitor the values of streams via a `monitor!` statement, for example on Line 6 of Listing 5. Whenever the `adder` actor emits a new value to its `out` stream, then because of the `monitor!` statement, a new `print!` message will arrive in the mailbox of the `Main` actor. When processed, this will lead to the invocation of the corresponding `print!` method on Line 7 that prints the output of the reactor to the console.

4 Introducing Poker

Poker is an open instrumentation platform for Stella programs. It visualises a running Stella program at the granularity of actors, reactors, and the streams that connect them. Programmers interact with Poker by hooking it up to a running Stella program, after which they can instrument said

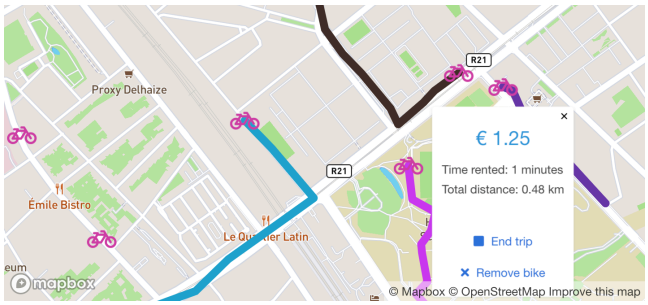

Figure 1. Screenshot of the instrumented Stella application.

program via a set of built-in *probes* that inspect various properties of the data that flows through the Stella program.

In the following sections we first introduce the Stella program that is used to exemplify the instrumentation, and afterwards we introduce various built-in probes that may help a programmer solve a bug, or gain extra insight into how the instrumented program operates at run-time.

4.1 A Stella Application

We demonstrate Poker by instrumenting the Stella application shown in Figure 1. The application simulates shared bicycles that float around a city for users to rent, and displays them on a map in real-time. Every bike is represented by an actor that continuously emits its location to a stream. Using this stream, three connected reactors are responsible for tracking bicycles' path (drawn via a coloured line), the time rented, and the current price of the rental which is calculated based on distance travelled and the elapsed time. While Stella is capable of distributing actors and reactors over a network, all actors and reactors giving rise to Figure 1 (as well as Poker) are running in the local browser.

Poker and its visualisation of our application at hand is shown in Figure 2. Each box represents an actor or reactor, and each arrow indicates a connection via a stream. In this case the application is tracking a single `Bike` actor (left), whose data flows through other actors and reactors. From left to right: `PathCalc` accumulates the travelled path, `PriceCalc` calculates the price, and `TripMonitor` aggregates info about the trip (journey) of a rented bike. The data from all trips (also other bikes) are collected by an actor called `TripManager`, and at the end the desired data of each rented bike reaches the `Main` actor (right) which modifies the GUI shown in Figure 1.

The blue buttons at the bottom of Figure 2 depict Poker's built-in probes, each of which directly corresponds to an actor behaviour in Stella. Clicking on a button creates a new (unconnected) box in the diagram. Users draw new arrows (via drag & drop) to connect the probes to existing streams: Incoming streams are connected to the left, and outgoing streams depart from the right. All changes are tentative until the user clicks the teal "commit changes" button to modify the instrumented application, after which the corresponding actors will be spawned and the streams connected.

4.2 Probe 1: Inspecting values

A developer notices a discrepancy in the calculation of the price of a rented bike. The price is established based on a fixed cost, the time rented, and the distance travelled. When the computed price is not as expected, then there must be a problem with the price calculation itself (`PriceCalc` in Figure 2), or with the accumulation of a bike's path (`PathCalc`), e.g., faulty location data produced by the bike.

4.2.1 Assessing the Problem with Poker. To inspect the values that are propagated via streams, Poker has a built-in `ValueProbe` that logs the values that are propagated via streams. This probe is effectively the "Hello World!" of Poker. Figure 3 depicts the addition of two such probes. The steps to add these probes are the following. (1) We clicked the blue `ValueProbe` button in Figure 2 two times, yielding two "unconnected" probes in the GUI. (2) We dragged a new arrow from the "output" of `PriceCalc` and `PathCalc` to the "input" of their respective probes. These new arrows are highlighted in Figure 3 for clarity. (3) We pressed "commit". Any values emitted by `PriceCalc` and `PathCalc` are now written to a log by the respective probes, which can be accessed by clicking the probe. A developer can use this log to debug the problem at hand.

4.2.2 Implementation. Probes are implemented in Stella. Due to their imperative nature (e.g., interactions with Poker's GUI) they are implemented as actor behaviours. For example, Listing 6 shows the implementation of `ValueProbe`. It defines a local field called `env` (Line 2), a constructor called `init` (Line 3), and a method called `log!` (Line 6).

The constructor arguments are provided by Poker when a user presses the "commit" button and the corresponding probes are spawned (by Poker). The first argument (`_env`) is an object of type `JSObjectProxy`, i.e., it is a JavaScript object that automatically wrapped by Stella's foreign function interface. Any method invocations on this object are forwarded by the Stella runtime to the wrapped JavaScript object, automatically wrapping/unwrapping Stella objects when needed. All following arguments to the constructor are expected to be `Stream` objects (i.e., the result of an `a.b` expression, see Section 3.4). The body of the constructor stores `_env` in its similarly named local field (Line 4) and starts monitoring the given stream (Line 5).

The `log!` method is invoked every time the monitored stream produces a new value (see Section 3.4). Its body invokes the `update!` method on `env` (Line 7), which (via Stella's foreign function interface) invokes a JavaScript function supplied by Poker. Its first argument `'text` indicates that Poker should store the given value `val` in a plain text log.

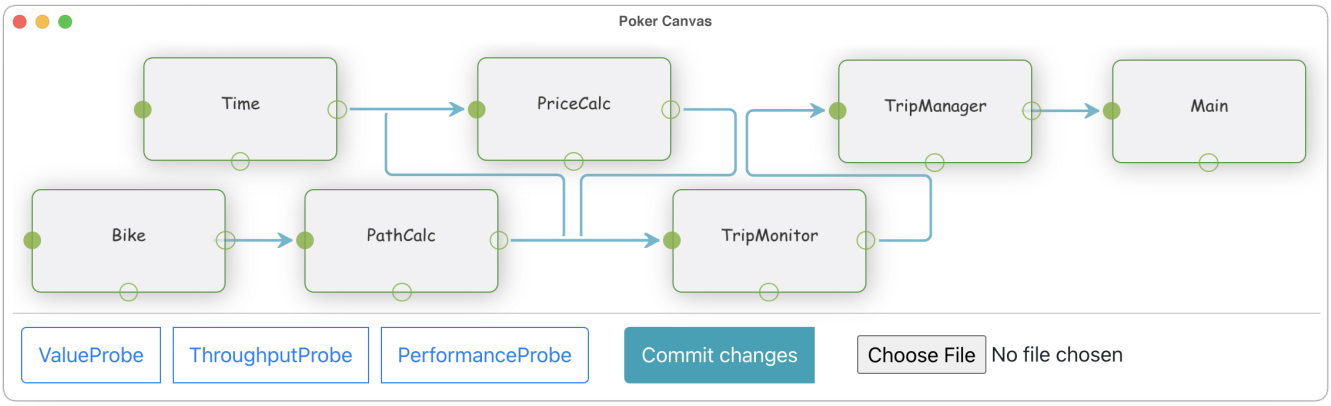

Figure 2. Screenshot of Poker.

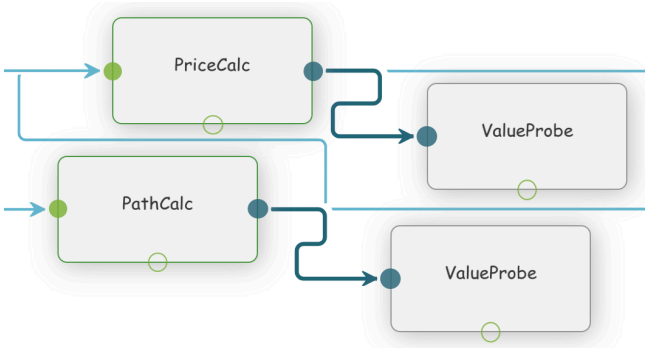

Figure 3. Adding 2 instances of a `ValueProbe`.

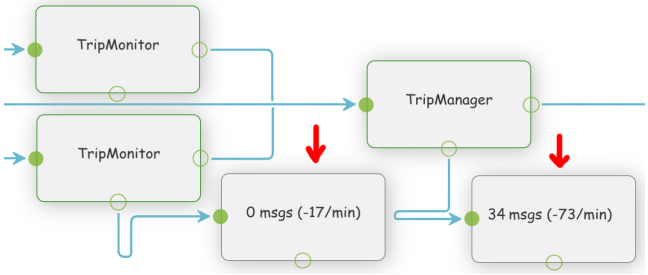

Figure 4. Adding 2 instances of a `PerformanceProbe`. Note that this figure depicts two instances of `TripMonitor` (and a third instance off-screen to the left) whose data is streamed to one instance of `TripManager`.

```
1 (def-actor ValueProbe
2   (def-fields env)
3   (def-constructor (init _env stream)
4     (set! env _env)
5     (monitor! stream 'log!))
6   (def-method (log! val)
7     (update! env 'text val)))
```

Listing 6. Stella implementation of `ValueProbe`

4.3 Probe 2: Measuring Performance

A developer notices that updates to the application input (e.g., a bike moves) take a long time to propagate through the application, for example, to update the GUI. This means there may be a bottleneck in the system that is unable to cope with high demand.

4.3.1 Assessing the Problem with Poker. A developer may try to find the bottleneck that causes the application to react slower than expected. Poker has a built-in probe called the `PerformanceProbe` that is used to assess the performance of an actor or reactor. Two such probes are shown in Figure 4 (highlighted via the red arrows). They are connected to the "bottom port" of `TripMonitor` and `TripManager` respectively because they measure properties about the actor or reactor itself, rather than the streams entering or leaving. Once connected, they show 2 metrics about the connected actor or reactor:

1. The size of their mailbox, i.e., the message backlog.
2. How fast are messages processed, measured in messages per minute.

For example, the rightmost node reads that there are currently 34 messages in the mailbox of `TripManager`, and that it processed 73 messages in the last minute. A large message backlog may suggest that the implementation of `TripManager` should be improved.

4.3.2 Implementation. The `PerformanceProbe` is implemented in Listing 7. This actor behaviour defines a couple of local fields (Line 2), an `init` constructor (Line 4) and 3 auxiliary methods (Line 10, 18 and 22). The main idea is that the probe continuously monitors the size of the mailbox of the "probee" (the one who is probed). This information, among other things, is published by every Stella actor and reactor via a `meta` stream that they export (which is used in the same way as any other stream). Besides tracking mailbox

```
1  (def-actor PerformanceProbe
2    (def-fields env size rate)
3
4    (def-constructor (init _env stream)
5      (set! env _env)
6      (set! size 0)
7      (set! rate 0)
8      (monitor! stream 'register-update!))
9
10   (def-method (register-update! stats)
11     (def new-size (get stats 'mailbox-size))
12     (when (< new-size size)
13       (set! rate (+ rate 1))
14       (send-after! #self 60000 'decr!))
15     (set! size new-size)
16     (send! #self 'update-gui!))
17
18   (def-method (decr!)
19     (set! rate (- rate 1))
20     (send! #self 'update-gui!))
21
22   (def-method (update-gui!)
23     (update! env 'inline (append "" size "
           msgs (-" rate "/min)"))))
```

Listing 7. Stella implementation of `PerformanceProbe`

size, the probe tracks a sliding window of the messages that were processed in the last minute.

The local fields on Line 2 are used to store the metrics: `size` will hold the last known size of the mailbox, and `rate` how many messages were processed in the last minute. Similar to `ValueProbe`, the `init` constructor accepts an argument `_env` to interact with Poker. Since the probe is connected in the GUI to the bottom port of the probee, the second constructor argument is the aforementioned `meta` stream. The body of the constructor initialises fields and monitors the given stream, invoking `register-update!` every time the stream emits a new value.

The `meta` stream emits a dictionary with various info such as the `mailbox-size`, retrieved on Line 11. Whenever the updated size is smaller than previously recorded in the `size` field, then the probee must have processed a message. The message is then counted towards the current rate of processed messages by incrementing `rate` (Line 13), and on Line 14 the probe schedules a `decr!` message to be sent to itself after 60000 milliseconds to decrease `rate` by 1 (the identifier `#self` refers to the current actor). Finally, the last known size is updated (Line 15), and the probe sends a message to itself to update the GUI.

Similar to `ValueProbe`, the GUI is updated by invoking an `update!` method on `env` (Line 23). In this case an `'inline` visualisation mode is selected, which will draw the string constructed via `append` directly in the box of the probe, as seen in Figure 4.

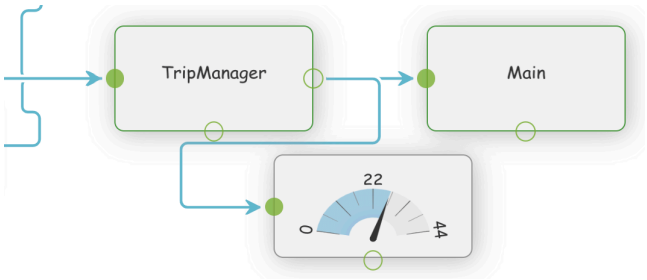

Figure 5. Adding an instance of a `ThroughputProbe`.

4.4 Probe 3: Measuring Throughput

The browser seems to exhibit a large CPU load related to rendering the map in Figure 1. This means that the number of updates to the GUI possibly exceeds what the application can handle.

4.4.1 Assessing the Problem with Poker. Poker can be used to assess whether the number of GUI update events is exceptionally high, or whether the implementation of the GUI is too slow to handle a normal load. This problem boils down to measuring the throughput of values of a stream, which is implemented by Poker's `ThroughputProbe`. Since the `Main` actor of the instrumented application is responsible for the GUI, in Figure 5 we attached the probe to its incoming stream. The gauge shows how many messages were propagated on the stream in the last minute.

4.4.2 Implementation. In Listing 8 we show the implementation of `ThroughputProbe`. It is implemented similarly to the previously discussed `PerformanceProbe`, because it also tracks a "sliding window" (via a counter) of values that are emitted to a given input stream. Concretely, due to the `monitor!` statement on Line 8, the `incr!` method is invoked every time the monitored stream publishes a value, and a `decr!` message is scheduled to be sent after 60000 milliseconds (Line 14). A notable difference is how the GUI is updated on Line 22. Here, Poker is instructed to visualise the data as a reactive gauge. Besides the current value of the gauge `ctr`, Poker also expects a lower bound and upper bound to determine its scale. In this case the bound is always between 0 and the highest throughput value ever seen, which is tracked in the local field `max-ctr`.

5 Poker as an Open Platform

To support programmers to get a better operational understanding of their program, Poker is designed as an *open* platform: Stella code can be used to extend the built-in set of probes. In other words, new probes with application-specific knowledge can be added. Concretely users of Poker can upload a new file of probes using the "Choose File" button in Figure 2 (bottom right). The probe definitions in that file are loaded without restarting the instrumented application, and

```
(def-actor ThroughputProbe
  (def-fields env ctr max-ctr)

  (def-constructor (init _env stream)
    (set! env _env)
    (set! ctr 0)
    (set! max-ctr 0)
    (monitor! stream 'incr!)
    (send! #self 'update-gui!))

  (def-method (incr! val)
    (set! ctr (+ ctr 1))
    (set! max-ctr (max ctr max-ctr))
    (send-after! #self 60000 'decr!)
    (send! #self 'update-gui!))

  (def-method (decr!)
    (set! ctr (- ctr 1))
    (send! #self 'update-gui!))

  (def-method (update-gui!)
    (update! env 'gauge ctr 0 max-ctr)))
```

Listing 8. Stella implementation of ThroughputProbe.

Poker's GUI is immediately extended with new (blue) buttons to add the probes. If the loaded file contains a probe definition that already exist (a name clash), then the new definition replaces the old behaviour, but it will only be applied to new probes (i.e., existing probes keep the old behaviour until they are explicitly replaced by new probes).

To exemplify the adding of new probes, we continue the example of Figure 5 where the throughput of a stream was too high, causing the GUI of the instrumented application to slow down due to high CPU load. If the implementation of the GUI is not the problem, then the situation can still be remedied by slowing down the rate of GUI updates. More specifically, the developer has determined that in this application GUI updates can be batched to reduce the number of times the map of Figure 1 is redrawn, thus reducing CPU load. They decide to implement an application-specific probe to experimentally determine an acceptable update rate.

5.1 Implementing New Probes

The implementation of probes must be conform to Poker's API. We briefly summarise its requirements.

Every probe must have a constructor called `init` that accepts at least one argument which, up to this point, we called `env`. As previously mentioned, this is an object of type `JSObjectProxy` that is provided by Poker when it spawns the probe, i.e., it is a Stella wrapper for a JavaScript object which is used by the probe to interact with Poker. All succeeding arguments to the constructor are expected to be objects of type `Stream` (i.e., the result of an `a.b` expression, see Section 3.4). They directly correspond to the "input ports" of a box in the visualisation. For example, since the

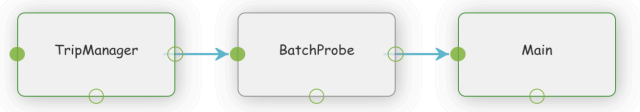

Figure 6. Batching GUI updates with BatchProbe.

```
(def-actor BatchProbe
  (def-stream batched-updates)
  (def-fields buffer frequency)

  (def-constructor (init env stream)
    // new: instantiate object-oriented class
    (set! buffer (new List))
    (set! frequency 1000) // 1 second
    (monitor! stream 'append-update!)
    (send-after! #self frequency 'flush!))

  (def-method (append-update! updates)
    (set! buffer (append buffer updates)))

  (def-method (flush!)
    (emit! batched-updates buffer)
    (set! buffer (new List))
    (send-after! #self frequency 'flush!)))
```

Listing 9. Stella implementation of BatchProbe.

ThroughputProbe of Listing 8 has 2 constructor arguments (1 of which is `env`), Poker draws the corresponding box in Figure 5 with 1 circle on the left to connect an arrow (a stream). While we have not shown it, multiple input streams are supported. They will appear as multiple inputs from top to bottom in the order of the constructor arguments. By connecting an arrow and pressing the "commit" button (see Figure 2), Poker spawns the probe with an `env` object and the connected streams as arguments.

To display values in the Poker GUI, probes must invoke the `update!` method on `env`. Its first argument is a display mode that determines how Poker should visualise the values. Supported modes are `'text` (write to log, see Section 4.2), `'inline` (show directly on the probe, see Section 4.3), and `'gauge` (draw dynamic gauge, see Section 4.4).

5.2 Batching GUI Updates

Given the above specification, the developer implements a new probe called `BatchProbe` that batches GUI updates. As shown in Figure 6 it is connected "inline" before GUI updates reach the `Main` actor. The user connected this probe by dragging & dropping the stream between `TripManager` and `Main` to the input port of `BatchProbe`, and by dragging a new arrow from the output port of `BatchProbe` to `Main`.

The implementation of `BatchProbe` is presented in Listing 9. The main idea is that all GUI updates are buffered in a list that is occasionally flushed to the GUI, in this case once every second. Many parts of the implementation are

similar to the built-in probes. However, a crucial difference is that `BatchProbe` impacts the application logic of the instrumented application: It intercepts a stream of GUI updates, and exports a new stream with less frequent (batched) updates. This output stream is declared on Line 2.

A `buffer` is initialised on Line 7 for storing the incoming GUI updates, and the frequency of updates is hard-coded to be every 1 second (Line 8)[4]. Every time the given input stream emits new GUI updates, these updates are appended to the stored buffer via the `append-update!` method on Line 12. The buffer is periodically flushed to the output stream via an asynchronous loop, where the probe sends itself `flush!` messages (Lines 10 and 18).

The output stream is automatically incorporated by Poker into the graphical representation of the probe (in Listing 9) via a circle on the right side of the probe, i.e., an "output port". By redirecting (via drag & drop) the arrow that used to connect `TripManager` and `Main`, behind the scenes Poker automatically performs the necessary work to seamlessly redirect streams in the instrumented application by interfacing directly with the Stella runtime.

6 Interaction Between Poker and Stella

The Stella interpreter was modified to expose some of its internal information required to implement a tool such as Poker. To aid the design of tools such as Poker for other (actor-based) reactive programming languages or frameworks, we briefly describe the kind of information exposed by Stella's interpreter.

In general, the language or framework should minimally support the following features, e.g., via metaprogramming, or in our case by modifying Stella's interpreter.

List (re)actors: Poker requires a list of every (re)actor in the program. In our case, Stella's interpreter notifies Poker every time a process is spawned or killed.

List stream dependencies: Every re(actor) should notify Poker whenever they establish or remove a subscription to a stream. This information is required to draw the data flow between (re)actors.

Spawn actors: To add probes, Poker should be able to spawn new actors with the desired behaviour.

Create new behaviour: To support the open nature of Poker, the language or framework should allow new actor behaviours to be added. E.g., to load (i.e., compile or interpret) new code at run-time.

Modify stream dependencies: One of the more difficult requirements is the ability to modify stream dependencies. When one (re)actor is streaming data to another, then the language or framework should support a mechanism to reroute this stream from its original source to a probe, and to route the probe's output stream to the original source.

[4]Note that, to experiment with the frequency of updates, a developer can easily modify the code and upload a new file without stopping the application.

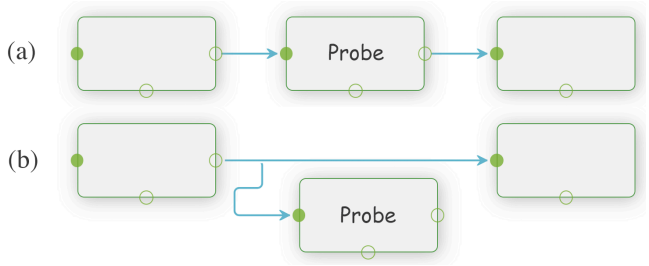

Figure 7. Series (a) vs. parallel (b) circuits

7 The Observer Effect

In the literature on reflection and meta-programming, a distinction is made between *introspection* and *intercession*. The former allows a meta-program to *refer* to values and structures of a running base program. The latter allows the meta-program to *replace* some of the values and structures of said program. Poker gives the developer the full intercessive power in the way Stella actors are linked to one another. This means that Poker probes may have wanted or unwanted effects on the execution of the program under instrumentation.

In physics, the idea that observing a system inevitable disturbs that system is known as **The Observer Effect**. Since Poker allows us to add and remove probes to and from a running Stella application, the question arises to what extent the observer effect plays a role when trying to understand a running system. We make a distinction between semantic disturbances and disturbances related to the run-time performance.

7.1 Qualitative Observer Effects

In electrical engineering a distinction is often made between *series circuits* and *parallel circuits* when connecting electrical components such as resistors, inductors and capacitors. This is illustrated in Figure 7.

The probes illustrated in Section 4 are wired into the DAG of the running application in a similar way. E.g., the `ValueProbe` probe presented in Section 4.2 was connected to the application in a parallel circuit. Hence this probe cannot incur a semantic observer effect on the running application. This is different for the `BatchProbe` presented in Section 5. This probe was connected to the application in a series circuit and can thus affect the values flowing through the program.

7.2 Quantitative Observer Effects (Benchmarking)

Apart from incurring a semantic effect on the running program, one might wonder what the cost is of adding probes to a system. When using series circuits, there will be an obvious cost that will depend on the complexity of the code sitting in the probe. More interesting is to know whether parallel circuits also have a performance hit. The extra work that needs to be done comes from the fact that a Stella (re)actor now

has to forward its output to more than just the application (re)actors that logically depend on that (re)actor.

In order to get a preliminary idea of the cost of wiring extra parallel probes in a running Stella application, we have conducted preliminary benchmarks that vary along the following dimensions:

1. **The number of times that the data is copied.** Since actors and reactors have no shared (mutable) state, messages between them are always sent by (deep) copy. Hence, adding an additional recipient (a probe) will slow down the overall performance of the system since the data needs to be copied more than once.
2. **The size of the data that is copied.** It is clear that an actor that emits primitive numbers will experience a smaller slowdown than an actor that emits "heavy" data structures that take longer to copy.

7.2.1 Benchmark Setup. Stella is implemented as a continuation passing style interpreter in TypeScript. We ran the experiments on Ubuntu 20.04.2 LTS with Node.js v14.16.0. While Stella is single-threaded, experiments were run on an AMD Ryzen™ Threadripper™ 3990X with 128GB of DDR4-3200 RAM (our benchmark only used around 1.6GB). All measurements were measured using Node.js' "performance measurement API", which implements the W3C recommendation for high resolution time with sub-millisecond precision [11].

The benchmarked application is simple: 1 actor repeatedly emits data to its stream, which is received by a varying number of `ValueProbe` actors (from Listing 6). We modified Stella's interpreter to measure the time it takes (on average) to execute the `emit!` statement of the data emitter.

1. To measure the impact of larger numbers of probes, we ran the benchmark multiple times with 1 to 64 probes (stepping the powers of 2).
2. To measure the impact of data size, we ran the benchmark with a producer of small data (primitive numbers) and a producer of larger data (a list of 1000 actor references).

Since the measured time of a single emission will be very small, we emitted the data 1,000,000 times. Total run-time of the benchmarks ranges between approximately 15 seconds (1 probe) and 10 minutes (64 probes) for the small data, and 30 seconds to 30 minutes for the larger data.

7.2.2 Results. The results of our benchmarks are graphed in Figure 8. The run-times using small data are depicted in blue, and those with large data in orange. As expected, the propagation times for both small and large data grow linearly with the number of probes, and copying larger data takes longer with more probes.

Considering the measured times are in the order of microseconds, attaching parallel probes has an *almost* (but not entirely) negligible observer effect on the application that is being instrumented. However, one can imagine using

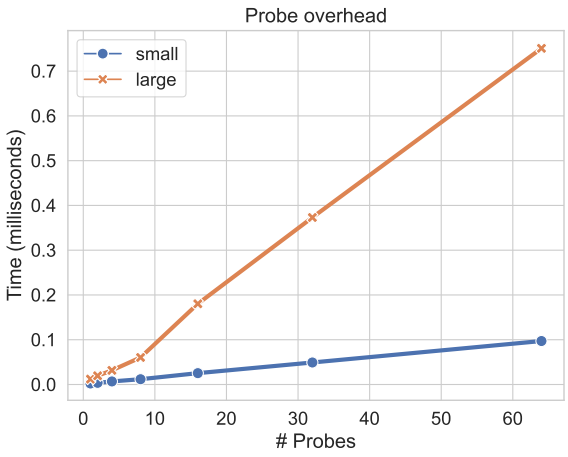

Figure 8. Benchmark results. Error bars = 95% confidence interval over 1,000,000 runs (drawn, but not visible due to their small size).

Poker to create rich dashboards by automatically attaching a whole range of parallel probes to various (if not all) actors and reactors in the application. Since the dashboard then indiscriminately adds many probes to the system, some of the measured stream values may contain large (i.e., slow to copy) data. In those cases the monitoring of Poker may unexpectedly impact the total performance or throughput of a system. To remedy these issues, approaches exist to pass data between actors without copying, e.g., via reference capabilities demonstrated by the Pony language [4].

8 Limitations and Future Work

8.1 Configuring Probes

The metrics collected by probes currently use hard-coded constants. For example, the `ThroughputProbe` always measures the throughput of a stream in values per minute. Currently developers have to upload a file with new probes to change these constants. It is conceivable to further extend Poker to allow probes to be configured even at run-time, e.g., by requiring probes to implement an additional method (e.g., `configure!`) such that Poker can send configuration messages to probes. A possible use case is a GUI slider that changes the reporting period of a `ThroughputProbe`.

8.2 Many-to-one Hierarchies

Poker is currently most suited to probe a single stream between two (re)actors. While probes with multiple input streams are supported (as long as the number of streams is fixed), sometimes situations arise where an actor or reactor is on the receiving end of a variable number of streams. For example, `TripManager` in Figure 2 (right) receives data from one stream for each rented bike in the application. Currently we do not support such many-to-one hierarchies where a

probe intercepts a variable number of the input streams. This seems to be challenging to implement from a GUI perspective, and also has its own challenges from the perspective of a probe's implementation when adding such probes in series-type circuits, since there currently is only one output stream. In analogy with electronics, we suspect this situation is similar to a multiplexer and demultiplexer.

8.3 Cause and Effect

When we initially experimented with Poker and its implementation, an appealing type of probe that perfectly fits within Poker's paradigm is a "latency probe": A probe that measures the latency between two parts of the system, i.e., the time it takes for a message to be processed between two selected points. However, this type of probe is currently very difficult to implement because it is not possible to track "cause and effect" between streams. More specifically, it is not possible to know that a certain input message (e.g., for an actor) gives rise to a certain output message published to its stream.

8.4 Complex Data Structures

In general, any Stella object can be passed from Stella to Poker (e.g., via the `update!` method used by probes). Simple Stella objects such as numbers, strings, dictionaries and arrays naturally map to a JavaScript equivalent. However, if no reasonable JavaScript equivalent exists (e.g., for complex data structures), Stella converts them to JSON before passing the objects to Poker. Concretely this means that Poker has no special mechanisms to visualise complex data other than the standard JavaScript tools, e.g., printing "prettified" JSON.

8.5 Beyond Stella

Poker targets actor-based reactive programs, which seems to limit its applicability in practice. However, the combination of reactive programming and actors has been noticed by others as well [22]. A notable example from industry is Akka Streams [19, Chapter 13], a framework based on the principles of "reactive streams" [1] built on top of the Akka actor framework [19] for Scala. Akka Streams allows developers to construct and compose "flows", which correspond to a DAG. Flows are executed by dedicated actors that are responsible for propagating values through the DAG. Thus, Akka Streams implements a model of Akka actors that communicate with Akka Streams "reactors". Given sufficient knowledge about metaprogramming in Akka, we believe it is conceivable to build a version of Poker for Akka Streams. The notion of debugging actor-based systems by means of a visualisation has been proposed before by [15].

9 Related Work

Throughout this paper we have discussed some related work for building visualisations of reactive systems. In this section we aim to more concretely distinguish Poker from those other works.

[21] describes a stop-the-world debugger with operations designed specifically for reactive programs. Beyond providing insight in the functional requirements of a reactive program, the debugger offers the means of measuring performance in a reactive system. They note that the typical approach of profiling the time a program spends executing a function is not as relevant in reactive programming. Instead, the fraction of the global update-time spent in a certain function is key to measuring performance in a reactive program. Poker does not provide this functionality, since the notion of a single global update phase does not exist in Stella: Each (re)actor handles updates independently. Hence, visualisation of non-functional requirements differs in two ways between [21] and Poker. First, Poker measures reactive performance as *messages per time* (i.e., throughput) instead of time per update. Second, since Poker is an open platform, it can be extended with probes that measure other metrics.

In addition to *measuring* the performance of a reactive program, there is also the aspect of managing the performance of the visualisation tool itself. On this front, the authors of [21] foresee some issues with the scalability of their system. To support larger reactive programs they provide a number of "countermeasures" to the scalability issues for their "always on" style of visualisation. The core idea behind their countermeasures is to reduce the number of nodes to visualise by pruning away parts of the reactive program DAG. In addition, they offer a search feature and show a thumbnail to help the user navigate the visualisation. Poker does not currently offer such countermeasures, limiting the size of the programs which can practically be visualised with Poker. However, Poker's approach does not suffer from these scalability issues to the same extent: Actors and reactors are, by nature, at a coarser granularity than the individual program statements in [21], and the number of Poker probes is not proportionate to the size of the input program. However, similar problems will occur in large programs with many actors and reactors.

[3] describes RxFiddle, a graphic debugger for RxJS. RxFiddle visualises a small RxJS program as a marble diagram. Unlike Poker, RxFiddle heavily focuses on the functional aspects of a reactive program: what concrete value is produced where. Visualisation of non-functional aspects is not possible with RxFiddle's marble diagrams. With respect to scalability, the authors of [3] note that their visualisation works best when the visualised application has a small number of events (<20), and is not suitable for higher volume data flow. As future work they propose handling high volume streams differently, e.g., by offering filtering features or watch expressions. Poker's approach would enable implementing those filters and watch expressions in Stella code, and exposing that code as new probes through the openness of the Poker platform.

[12] describes the Vega debugger. This tool displays events produced inside a reactive program as rows of time series data, or plots them as, e.g., timelines of values. The visualisation of the functional data is completely detached from a visualisation of the reactive DAG structure. Visualisation of non-functional aspects is not supported in the Vega debugger.

[17] describes how a platform like Poker could be used not only for gaining insights, but also for live code manipulation to handle the insights gained. The work appears to be conceived mostly for experimenting with changes: Input values as well as the reactive structure can be modified, and time-travel debugging is supported to inspect how the changes impact the program's behaviour. In Poker we made the deliberate decision that the definitions of probes must be stored in a file, loaded into the application through the open platform as described in Section 5. The decision to load from files prevents us from offering the same degree of dynamism as [17], but gains us reproducibility.

Since Poker makes it possible to add code to a running program and modify its behaviour, Poker could be seen as a Live Programming Environment [14] or as a visual programming language or tool [18]. However, the goal of Poker is not to serve as a visual programming environment (or IDE) for Stella. A live or visual programming tool for Stella has to tackle additional concerns which are not tackled by Poker, e.g., viewing and modifying the code of existing (re)actors, hot code reloading, and persisting changes made via the tool to the underlying codebase.

10 Conclusion

We presented Poker, a visual instrumentation platform for reactive programs written in Stella. The main novelty of Poker is that: (a) Its probes can be transparently wired into a running program. (b) A probe can measure a functional or non-functional requirement of a particular stream. (c) Probes are programmed in Stella itself, and can be added to Poker without stopping the instrumented application. As such, Poker can form the basis for debuggers for perpetually running reactive platforms.

While the tool Poker is specifically aimed at Stella applications, the concepts and mechanisms are more broadly applicable. More specifically, Poker has demonstrated a new mode of interaction with a reactive program that enriches the toolbox of reactive programmers with which they can collect both functional and non-functional metrics about the reactive program. By allowing new probes to be programmed and added at run-time, programmers can gain a better understanding of their programs.

Acknowledgments

Sam Van den Vonder is funded by the Research Foundation - Flanders (FWO) under grant number 1S95318N. Thierry Renaux is funded by the Flanders Innovation & Entrepreneurship (VLAIO) "Cybersecurity Initiative Flanders" program.

References

[1] [n.d.]. Reactive Streams. http://web.archive.org/web/20191009093755/https://www.reactive-streams.org/ Accessed 2019-10-09.

[2] Manuel Alabor and Markus Stolze. 2020. Debugging of RxJS-Based Applications. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems* (Virtual, USA) *(REBLS 2020)*. Association for Computing Machinery, New York, NY, USA, 15–24. https://doi.org/10.1145/3427763.3428313

[3] Herman Banken, Erik Meijer, and Georgios Gousios. 2018. Debugging Data Flows in Reactive Programs. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018* (Gothenburg, Sweden) *(ICSE '18)*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). Association for Computing Machinery, New York, NY, USA, 752–763. https://doi.org/10.1145/3180155.3180156

[4] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE! 2015, Pittsburgh, PA, USA, October 26, 2015*, Elisa Gonzalez Boix, Philipp Haller, Alessandro Ricci, and Carlos Varela (Eds.). Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2824815.2824816

[5] Gregory H. Cooper and Shriram Krishnamurthi. 2006. Embedding Dynamic Dataflow in a Call-by-Value Language. In *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3924)*, Peter Sestoft (Ed.). Springer-Verlag, Berlin, Heidelberg, 294–308. https://doi.org/10.1007/11693024_20

[6] Massimiliano Sassoli de Bianchi. 2013. The observer effect. *Foundations of science* 18, 2 (2013), 213–243. https://doi.org/10.1007/s10699-012-9298-3

[7] Sam Van den Vonder, Thierry Renaux, Bjarno Oeyen, Joeri De Koster, and Wolfgang De Meuter. 2020. Tackling the Awkward Squad for Reactive Programming: The Actor-Reactor Model. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 19:1–19:29. https://doi.org/10.4230/LIPIcs.ECOOP.2020.19

[8] Jonathan Edwards. 2009. *Coherent Reaction.* Technical Report MIT-CSAIL-TR-2009-024. Massachusetts Institute of Technology, Computer Science and Artificial Intelligence Laboratory, Cambridge, 02139 Massachusetts, USA. http://web.archive.org/web/20181103183154/http://dspace.mit.edu/bitstream/handle/1721.1/45563/MIT-CSAIL-TR-2009-024.pdf?sequence=1

[9] Martin Glinz. 2007. On Non-Functional Requirements. In *15th IEEE International Requirements Engineering Conference, RE 2007, October 15-19th, 2007, New Delhi, India.* IEEE Computer Society, Washington, D.C., United States, 21–26. https://doi.org/10.1109/RE.2007.45

[10] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley Longman Publishing Co., Inc., Boston, Massachusetts, USA. ISBN: 978-0-201-11371-6.

[11] Ilya Grigorik. 2019. *High Resolution Time Level 2.* W3C Recommendation. W3C. https://web.archive.org/web/20210325154905/https://www.w3.org/TR/2019/REC-hr-time-2-20191121/ Accessed: 2021-03-25.

[12] Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. 2016. Visual Debugging Techniques for Reactive Data Visualization. *Computer*

Graphics Forum 35, 3 (2016), 271–280. https://doi.org/10.1111/cgf.12903 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.12903

[13] Jay Kreps, Neha Narkhede, and Jun Rao. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB, 2011, Athens, Greece*, Vol. 11. Association for Computing Machinery, New York, NY, USA, 1–7.

[14] Louis Mandel and Florence Plateau. 2009. Interactive Programming of Reactive Systems. *Electron. Notes Theor. Comput. Sci.* 238, 1 (2009), 21–36. https://doi.org/10.1016/j.entcs.2008.01.004

[15] Aman Shankar Mathur, Burcu Kulahcioglu Ozkan, and Rupak Majumdar. 2018. IDeA: An Immersive Debugger for Actors. In *Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang* (St. Louis, MO, USA) *(Erlang 2018)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3239332.3242762

[16] Leo A. Meyerovich, Arjun Guha, Jacob P. Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. 2009. Flapjax: a programming language for Ajax applications. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, Shail Arora and Gary T. Leavens (Eds.). Association for Computing Machinery, New York, NY, USA, 1–20. https://doi.org/10.1145/1640089.1640091

[17] Ragnar Mogk, Pascal Weisenburger, Julian Haas, David Richter, Guido Salvaneschi, and Mira Mezini. 2018. From Debugging Towards Live Tuning of Reactive Applications. In *2018 LIVE Programming Workshop. LIVE*, Vol. 18.

[18] Mitchel Resnick, John H. Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay S. Silver, Brian Silverman, and Yasmin B. Kafai. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (Nov. 2009), 60–67. https://doi.org/10.1145/1592761.1592779

[19] Raymond Roestenburg, Rob Bakker, and Rob Williams. 2016. *Akka in action* (1 ed.). Manning Publications Co., Shelter Island, New York, United States. ISBN: 978-1-61729-101-2.

[20] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. 2014. REScala: bridging between object-oriented and functional style in reactive applications. In *13th International Conference on Modularity, MODULARITY '14, Lugano, Switzerland, April 22-26, 2014*, Walter Binder, Erik Ernst, Achille Peternier, and Robert Hirschfeld (Eds.). Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/2577080.2577083

[21] Guido Salvaneschi and Mira Mezini. 2016. Debugging for Reactive Programming. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). Association for Computing Machinery, New York, NY, USA, 796–807. https://doi.org/10.1145/2884781.2884815

[22] Kazuhiro Shibanai and Takuo Watanabe. 2018. Distributed Functional Reactive Programming on Actor-Based Runtime. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control* (Boston, MA, USA) *(AGERE 2018)*, Joeri De Koster, Federico Bergenti, and Juliana Franco (Eds.). Association for Computing Machinery, New York, NY, USA, 13–22. https://doi.org/10.1145/3281366.3281370

[23] David M. Ungar and Randall B. Smith. 2007. Self. In *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007*, Barbara G. Ryder and Brent Hailpern (Eds.). Association for Computing Machinery, New York, NY, USA, 1–50. https://doi.org/10.1145/1238844.1238853