School of Information and
Communication Technology

Creating the Conveyerl visualisation tool for Erlang programs

Johan Tjelldén and Erik Ylipää

Supervisor: Joe Armstrong (Ericsson AB)
Examiner: Christian Schulte (School of ICT)

Degree Project in Information
and Software Systems
First Level, 15.0 HEC
Stockholm, Sweden 2012

TRITA-ICT-EX-2012:142

Every problem of computer science can be solved by another level of indirection, except the problem of too many levels of indirection.
— Unattributed

Sammandrag

Inom de flesta ingenjörsområden — forutom programvaruutveckling — har ingenjören möjligheten att fysiskt kunna skåda resultatet av sitt arbete. En ingengör som jobbar med en bro kommer så småningom att kunna se frukten av sitt arbete som en fysisk, och ofta magnifik konstruktion. Systemen som programvaruutvecklare skapar är inte mindre magnifika i deras komplexitet, men de är helt abstrakta.

Det är en fascinerande idé att kunna att *se* det mjukvarusystem vi har skapat. Vi tror att mjukvara kan vara vacker, inte bara på algoritmers, satsers eller uttrycks nivå, utan vacker på systemetnivå.

Denna tro utgjorde starten för vårt kandidatexjobb. Vårt mål var att visualisera Erlangprogram. För att göra detta ville vi skapa ett program som kan analysera spårningen ifrån en körning av ett Erlangprogram, och skapa en visualisering om hur processerna från det spårade programmet interagerar med varandra.

Mjukvaran som vi skapade har en framdel som körs i webbläsaren och en bakdel som körs i Erlangs virituellamaskin. Framdelen är skriven i JavaScript och använder tekniker som WebGL, WebSockets och SVG. Bakdelen är skriven i Erlang och sköter all spårning och grundläggande analys av programmet. Bakdelen har en inbyggd webbserver som används för kommunikationen mellan de två delarna.

I den här rapporten tar vi upp lite bakgrund om Erlang och mjukvaruvisualisering. Vi beskriver två alternativa användningsmöjligheter som har drivit utvecklingen vår mjukvara. Vi tittar på några designval vi har gjort under utvecklingen och arkitekturen av den färdiga mjukvaran. Vi avslutar med en utvärdering och med slutsatser.

Abstract

Most fields of engineering — save software engineering — have the advantage of being able to physically look at the works of their labour. An engineer working to create a bridge will eventually be able to see the fruits of his work as a physical, and often awe-inspiring construct. The systems which software engineers create are no less magnificent in their complexity, but they are completely abstract.

It is a fascinating idea to be able to *see* the software system we have created. We believe that software can be beautiful, not just on the level of algorithms, statements or expressions, but beautiful on the system level.

This belief was the start of our degree project. Our goal was to visualise Erlang programs. To do this, we wanted to create a program capable of analysing the trace produced by running an Erlang program, and create a visualisation of how the processes of the traced program interact with each other.

The software we created has a front-end running in a web browser and a back-end running on the Erlang virtual machine. The front-end is written in JavaScript and uses technologies like WebGL, WebSockets and SVG. The back-end is written in Erlang and does the sampling and basic analysis of Erlang programs. The back-end has an embedded web server which it uses to communicate with the front-end.

In this report we cover some background information about Erlang and software visualisation. We describe two use-cases which we used in our development efforts. We give an overview of the software and it's implementation details. We conclude with an evaluation and conclusions.

Contents

1 Introduction

The goal of this project has been to create a tool for visualising Erlang programs in order to assist teachers and students in the process of learning Erlang. To do this we have created the Conveyerl visualisation tool. It samples the trace of an Erlang program. After a program has been sampled, Conveyerl does a post-mortem analysis of the trace and visualise the result in a web browser.

We wanted to create a system which gives a clear and attractive visualisation of what is happening in an Erlang program. In order to achieve this we focused on the visualisation of processes and messages. We wanted our visualisation to be simple and easily extensible. We also wanted the visualisation to be useful as an aid in lectures, and have extended the software to allow teachers to define presentations based on the visualisation.

The field of software visualisation tries to help us understand the complexity of computer programs, using visual and auditory aids. Imagine that the instructions of a program are like the intricate streets of a city. Software visualisation is the attempt to produce a map over the twisting paths, to help us navigate through the complexity. Unlike the map of a city, software visualisation has to make choices of what to visualise and how to do it.

An important starting point for this projects was the question *What does a computer program look like?*

Other fields of engineering have the advantage of being able to physically look at the works of their labour. An engineer working to create a bridge will eventually be able to see the fruits of his work as a physical, and often awe-inspiring construct. The systems software engineers create are no less magnificent in their complexity, but they are completely abstract.

We believe it is a fascinating idea to be able to *see* the software system we have created. We believe that software can be beautiful, not just on the level of functions or statements, but beautiful on the system level. It is this beauty we would like to show to it's creator.

In this report, we present the Conveyerl visualisation tool and describe it's development. In section 2, we describe the Erlang programming language, software visualisation, web programming and general concepts referred to later in the report. Section 3 describes the use cases which served as design criteria. Section 4 gives an overview of Conveyerl and how it is used to sample and visualise a program. In section 5 we describe the architecture and technical details of our software. Section 6 discusses the result of Conveyerl in relation to the use cases. We also describe some of the choices we made during the development. In section 7 we describe related work. Section 8 presents ideas for future work. In section 9 we draw our conclusions.

2 Background

2.1 Erlang

Erlang is a programming language created at Ericsson in the 1980s and was released under an open source licence in 1998. It is a functional language with a strong focus on concurrency and high availability.

Programs in Erlang are organised in modules. A module is a collection of function definitions. Functions which are explicitly *exported* from the module are available from the outside, while those not exported are private and only available for calls from inside the module.

2.1.1 Concurrency model

Erlang's concurrency model is based on lightweight processes. A process is an isolated thread of execution with it's own isolated state. In a single node, millions of such processes can coexist. Processes interact with each other by sending messages. This design maps intuitively to how we as humans understand concurrency [2].

Every process in Erlang is associated with a Process Identifier (PID for short). A PID is automatically generated for a process when it is spawned. To communicate with a process it's PID is used as an address. Processes can register themselves under a global name, in which case the name can be used instead of the PID.

Communication is asynchronous, which means that as soon as a process has sent a message it continues with it's own execution. When a message is sent to another process it's placed in a mailbox which the receiving process checks when it evaluates a `receive` expression.

2.1.2 Process relationships

Erlang is designed for writing fault tolerant and robust systems. To simplify failure handling of processes, the programmer can create relations between processes. Processes can be *linked*, which means that if one of the processes exits, the other will be sent an `EXIT` signal. This way, processes which depend on each other can be linked together and stopped as a whole. Processes can also *monitor* other processes, in which case they receive signals when a monitored process exits. This is used to create processes which supervises others to automatically handle partial failures of the system.

2.1.3 Tracing and debugging

Erlang has a set of built in functions (BIFs for short) for tracing the execution of programs on a per process level. The function `erlang:trace/3` is used to enable or disable tracing. A process is set up as a receiver of trace messages and will continuously get information about the other processes as messages. Properties which can be traced are:

- The sending and receiving of messages.
- Function calls and returns. *Trace patterns* can be defined to filter what function calls will be traced.
- Exceptions.
- Process spawn and exit events.
- Process links and unlinks.
- Process registering and unregistering a name.
- Processes being context switched in and out of execution.
- Garbage collection.

2.2 Software visualisation

Software visualisation creates visual aids for conveying information about software systems. The field is generally broken down into two main categories [24]:

Algorithm visualisation uses graphics and animations to show how algorithms work. They are often used in education, while teaching the fundamentals of computer science [20]. Algorithm visualisation often focuses on visualising the general behaviour of an algorithm on a high level of abstraction.

Program visualisation delves into the more detailed behaviour of computer programs, on a low level of abstraction. It concerns itself with actual program code, data structures and the execution of instructions.

2.3 Publish/Subscribe Design Pattern

Design patterns are formulated "best practice" solutions for commonly occurring problems in software engineering. The concept is originally attributed to the architect Christopher Alexander [1]. The concept was applied to software engineering by Kent Beck and Ward Cunningham, later popularised by the so called "gang of four" [11].

The publish/subscribe design pattern is a system for handling communication between parts of a system. The pattern defines *channels*, which different entities in the system can subscribe to. When events are published to a channel, all subscribers are notified. The pattern introduces a level of indirection between publishers and subscribes of events, and makes them independent of each other.

2.4 Graph drawing

Graphs play a major role in a lot of scientific fields (Computer Science in particular), many methods to automatically create pictures of graphs have been developed. Di Battista et. al. provides a thourough bibliography of the subject [12].

A commonly used approach is a force based layout, where edges between nodes are treated as springs, and all nodes are given a repellent force towards each other. The springs will cause connected nodes to move towards each other, while the repellent force will push unconnected nodes apart. This system is simulated iteratively until it stabilises. The result will be a layout of the nodes where connected nodes are close to each others.

2.5 HTML and CSS

Hyper Text Markup Language (HTML) is the standardised language for structuring content for a web browser. It is based on nestable element tags, which is used to create a tree-structured document. HTML is often used together with Cascading Style Sheets (CSS). CSS is a language for describing the layout and style properties of the elements in the HTML document. A new version of HTML is in the process of being standardised, and introduces a number of new features [26].

2.6 JavaScript and the Document Object Model

JavaScript is a object-oriented, dynamically typed scripting language. It was originally designed for creating client side interaction in web pages. When used in a web browser, it is closely tied to the *Document Object Model* (DOM), which is a platform- and language-neutral model for programatically interfacing with documents [25]. It is the model used by JavaScript to access and modify content of a HTML document in a web browser.

2.7 WebGL

WebGL is an API for creating hardware accelerated 3D graphics in a web browser [17]. The standard is based on the OpenGL ES 2.0 API, which was originally designed for embedded systems such as mobile phones. WebGL extends the HTML5 specification and provides a 3D graphics rendering context for the HTML Canvas element.

The WebGL API is of a low level, and a number of libraries exist which makes it easier to use. Conveyerl uses the library Three.js as a framework for 3D graphics.

WebGL is supported in Mozilla Firefox from version 4.0 and Google Chrome from version 9.0 . Apple's Safari version 5.1 and Opera version 12 had has experimental support which is disabled by default. Microsoft

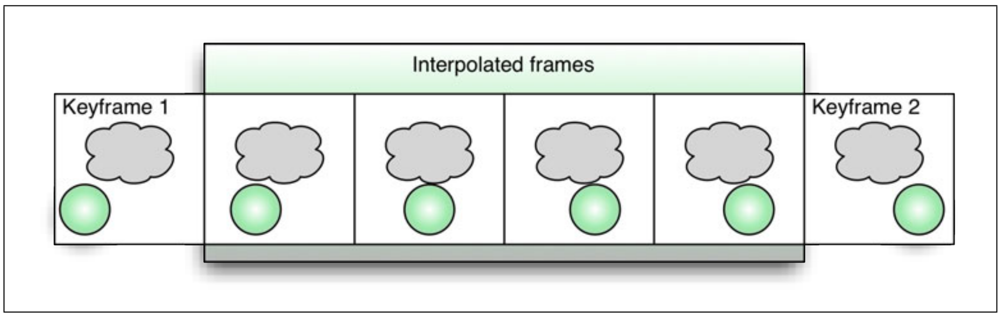


Figure 1: Key frames

Internet explorer has no support, and no plans have been announced of it being implemented in the future [18].

2.8 Computer graphics animations

Many computer animation techniques uses so called *key frames* to define the state of the graphics at a certain time (see figure 1). Key frames in classical animation were drawn by the lead artist to describe a scene. They often pictured the extremes of an animation. An assistant would take the key frames and fill in the frames in between them to create smooth transitions.

The same concept is used in computer animation, but the laborious work of the assistant is taken over by the computer which calculates positions for the animated objects at times in between the key frames, using different kinds of interpolations.

2.9 Web page animations

There are two ways of animating elements on a web page. One way is by the use of CSS animations which is a new feature in modern browsers. The other is by using JavaScript to change attributes over time.

CSS animations gives browsers a way to accelerate the animations on supported hardware. A major limitation is that it only works with the CSS style attributes of HTML elements. It can not be used to animate WebGL objects.

There are many JavaScript libraries for animation. Many of the most popular ones are not based on key frame animations, but instead uses chains of callback functions with timers to create complex animations. This limits the ability to pause and skip to different times in the animations. Another common limitation is that in general they are written solely for animating properties of DOM elements.

2.10 Asynchronous web communication

There are two general techniques for exchanging data between the web browser and web server without reloading the page:

AJAX Is an acronym for *Asynchronous JavaScript And XML*. It's a diverse collection of techniques for using JavaScript to communicate with the server in the background and manipulate the content of web pages without reloading the page. Ajax uses standard HTTP requests such as POST and GET to communicate with the web server. This introduces a header overhead, which becomes more significant if a lot of small messages are exchanged between the server and browser. To enable a server to push data to the browser, a technique called long polling is often used. With long polling, the browser makes a new request to the server whenever it gets a response. The server waits with responding until it has new data to send to the browser.

WebSockets A web socket gives the browser a direct stream-based connection to the server. The technology is still in the process of being standardised. WebSockets creates a communication channel which resembles a TCP socket, which can be used for arbitrary two way communication between the web browser and server. It removes a lot of the shortcomings of AJAX, such as HTTP header overhead and the need for long polling [19].

2.11 Data Exchange Formats

To exchange data between different representations (eg. Erlang terms to JavaScript values) a way of encoding and serialising the data is needed. The following formats were used in Conveyerl:

2.11.1 BERT

BERT is a binary data format based on how Erlang encodes its terms with the `term_to_binary/1` BIF, the external term format [9].

The format makes it easy to serialise Erlang data, and there are many libraries for encoding and decoding data to/from BERT. The objects act as representations for Erlang terms, and are annotated with the type (e.g. tuple) they represent.

2.11.2 JSON

JSON (JavaScript Object Notation) is a data format which is derived from how JavaScript encodes its data structures [7]. It provides an almost one-to-one mapping from JavaScript objects to a serialised character representation.

It is often used in web application together with Ajax to send and receive data between a web browser and server.

There are libraries for doing JSON encoding/decoding for most major programming languages. In Erlang there are a couple of different libraries, none of them official.

3 Use cases

Our goal was to create a visualisation tool for use in education. We defined design criterias and divided them into two different use cases: a student and a teacher perspective.

3.0.3 Student use case

The main purpose of the tool was to help students when learning how to program Erlang. We believed a visualisation could be a good aid when they start to understand how the Erlang programming model works, with processes interacting through messages. We also believed it could be useful for understanding the principles of distributed systems.

Interactivity For a student to be able to explore a program, the visualisation should be interactive [24]. A user should be able to interact with it to discover what happened in the program. Examining what the program is doing at different times was something we considered important, and the user should be able to easily navigate the time of the visualisation.

Ease of installing Installing the software should be easy, and require a minimal amount of extra steps. One way to simplify installation is by reducing the number of dependencies. The visualisation should be independent of operating system, ideally supporting the operating systems where Erlang is available.

Automated visualisation To simplify the usage of the tool, we wanted the visualisation to be automated. The user should have to do as little as possible to get to the visualisation from the analysed program. The visualisation should be able to visualise Erlang programs without the user having to change the program.

Beauty An important inspiration for the visualisation was the beauty of software systems. We wanted the visualisation to be attractive and pleasing. It should give the viewer a sense of wonder.

3.0.4 Teacher use case

The teacher use case is an extension to the requirements of the student use case. With the teacher use case, we focus mainly on the use of software visualisation as an visual aid for lectures. To use the the tool in lectures the teacher should be able to create presentations by tuning properties of the visualisation and add information to it. The following is a set of features we used as design criteria for a presentation tool:

Positioning processes The user should be able to edit the layout of processes at different times.

Positioning the view The user should be able to define what the visualisation is displaying at different times.

Breakpoints The user should be able to define *breakpoints* where the presentation should pause and await interaction before it continues playing.

Source code display To emphasize the connection between the source code of the program and the events of a run, the user should be able to display the source code and define highlights at different times to connect it to the visualised program.

Annotations The user should be able to annotate the visualisation with labels and text, and decide when and where they will appear and disappear.

Store and load presentations The tool has to be able to store and load presentations which a user has created.

Distribute visualisations The teacher should have a simple way of distributing presentations to students, allowing them to view and experiment on their own.

4 Overview

In this section we give an overview of Conveyerl. We give a brief description of its architecture followed by a description of its features and usage.

4.1 Architecture

The program is divided into two general parts: a back-end doing the sampling of programs and a front-end used for the visualisation. The back-end is controlled through an Erlang shell, while interaction with the front-end is done in a web browser. The term we used for a sampled program was a *run*. This term is used for all the data associated with one particular recording such as the trace data, the analysed data and the presentations.

Conveyerl creates a *post-mortem visualisation*, which means that the visualisation is created from a previously recorded sample of a program. The opposite is *live visualisation* where a visualisation is created continuously from a running program. Conveyerl does not support live visualisation.

A detailed description of the architecture and implementation details of Conveyerl is given in section 5.

4.2 Visualised properties

Conveyerl visualises the following properties of Erlang programs:

Process origin All processes are spawned with a specific start function. Conveyerl differentiates between processes depending on which function they were spawned with, giving processes with the same origin the same visualisation.

Process state Conveyerl visualises the running state of a process, if they are context switched in or out.

Life ranges Conveyerl visualises the life ranges of a process, it's spawn and exit events, and shows which process is the spawner.

Relationships Conveyerl visualises which processes are linked.

Messages Conveyerl visualises messages and their contents. It also shows which processes are the sender and receiver.

4.3 The visualisation

The front-end uses 3D graphics to visualise the sampled program. The visualisation works like a movie, where the trace of the program can be played like a recording of events by a video camera.

We wanted the visualisation to be attractive and recognisable. We considered astronomical imagery to be awe-inspiring, and wanted our visualisation to capture some of their sublime beauty. The two primary graphical entities are processes and messages. The program is visualised as a star system or a star cluster, where the processes are the stars. They shine in different hues and brightness depending on their properties at different times. Messages sent between them are visualised as projectiles.

4.3.1 Process layout

The layout of processes are used as a part of the visualisation. Conveyerl uses a force based graph layout to position processes. They are positioned according to which other processes they communicate with. The layout defines repelling and attracting forces between processes. All processes repel each other and the more they communicate the stronger the attracting force between them becomes.

4.3.2 Processes

Processes are visualised as stars. They have different colours and brightnesses depending on their origin and state (described below).

4.3.3 Process origin

To quickly give and idea of which function a process was spawned with, we assign colours to the processes. Processes spawned with the same function have the same colour. This way a user can get a quick idea of which processes make up different parts of a system.

4.3.4 Process state

To visualise the running state of processes we use the brightness of their colour. When a process gets context switched in it is set to maximum brightness. When it gets context switched out it's set to a lower brightness.

4.3.5 Life ranges

The life ranges of processes are determined by when they spawn and exit. Spawn events are visualised as a sphere travelling from the spawner to where the process is spawned. The sphere is the same colour as the spawned process. Exit events are shown as an explosion in the colour of the exiting process.

```
 Eshell V5.9  (abort with ^G)
1> conveyerl:start().
<0.35.0>
2> conveyerl:start_sampling(ring, test, []).
<0.112.0>

[...]

3> conveyerl:stop_sampling().
{stop_sampling,default_naming}
```

Figure 2: Example usage of the Erlang sampling interface

4.3.6 Relationships

Link relationships are shown as lines connecting processes. The line is coloured based on the processes which it links, with a gradient from one to the other.

4.3.7 Messages

Messages are visualised as projectiles going from the sender to the receiver. They are conical and aligned along the path they travel to clearly show their direction when the visualisation is paused.

4.4 Usage

4.4.1 Back-end usage

The user adds the program which should be analysed to the Conveyerl directory structure, by copying or creating a file system link. The user then runs `conveyerl:start/0` in an Erlang shell. This starts the back-end systems, including the web server.

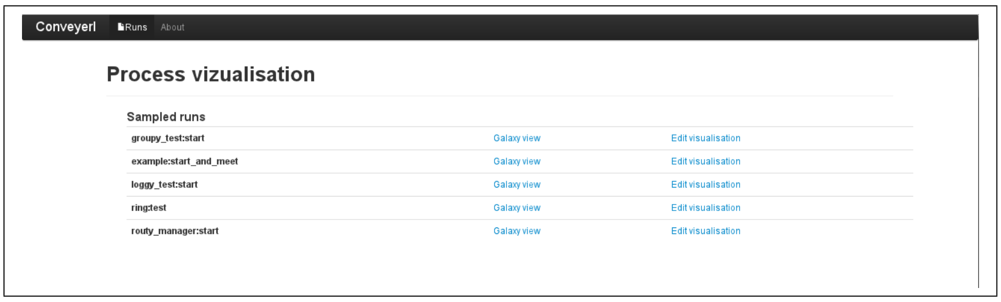
Once the back-end is started, a user can interact either through the shell or via a web browser. Interaction through the shell is mainly for sampling programs, exporting sample data and stopping the system. The web browser is used for viewing the sampled run as a visualisation.

4.4.2 Sampling of a program

To start a sampling, the user runs `conveyerl:start_sample(Module, Function, Arguments)`. When the user wishes to end the sampling, the function `conveyerl:stop_sampling/0` is run. See figure 2 for an example.

This creates a new sample-file in the `runs` directory. The file is a binary dump of the traces and process-info data gathered during the sampling.

4.4.3 Interacting with the visualisation



Process vizualisation

Sampled runs

groupy_test:start	Galaxy view	Edit visualisation
example:start_and_meet	Galaxy view	Edit visualisation
loggy_test:start	Galaxy view	Edit visualisation
ring:test	Galaxy view	Edit visualisation
routy_manager:start	Galaxy view	Edit visualisation

Figure 3: The listing of runs

To visualise the sampled program, the user connects to the web server with a browser (the default address is `http://localhost:8080`). A list of all performed samplings will be displayed. The user can choose either the main visualisation view, called *the galaxy view* or edit the visualisation as a presentation in *the editor view.*

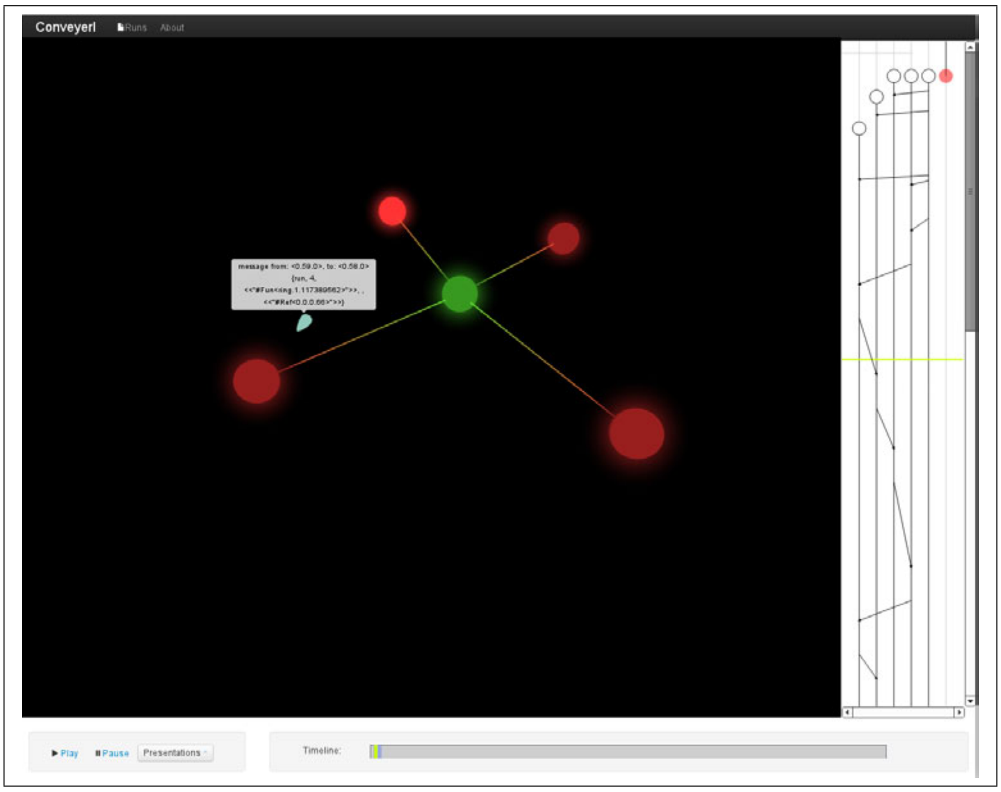
4.4.4 The galaxy view

The main view concentrates on the processes. It's divided into three main areas; the galaxy view, the event graph and the controls. See figure 4.

The galaxy view is the main visualisation. It's an interactive 3D view which can be zoomed, panned and rotated to view different processes of the sampled program. When the user hovers the mouse pointer over a process or message, the visualisation displays a tool tip with information about the object. For processes, the PID and spawn functions are displayed. For messages, the sender, receiver and payload are displayed.

Conveyerl provides different ways for a user to navigate the time of the visualisation. The event graph displays normalised time (see section 5.6), were periods of idleness are compressed, and eventful periods are expanded. The controls has a *timeline* which displays the time as real time, and most events are clustered with long gaps of idleness between them. Events are displayed as vertical bars.

The event graph is a sequence diagram which shows the processes of the program and the messages sent between them. It shows processes as vertical lines plotted over time. Messages are shown as arrows going between the process lines. The event graph is based on the compressed time described in section 5.6. The graph works like a time navigation tool and can be clicked in order to jump to different times.



```
message from: <0.59.0>, to: <0.58.0>
{run, 4,
<<"#Fun<ing.1.117389562>">>, ,
<<"#Ref<0.0.0.66>">>}
```

Figure 4: The galaxy view, Conveyerl main visualisation view. The sidebar to the right shows the event graph, and the bottom area shows the controls. The main area is the 3D-graphics visualisation of the running program.

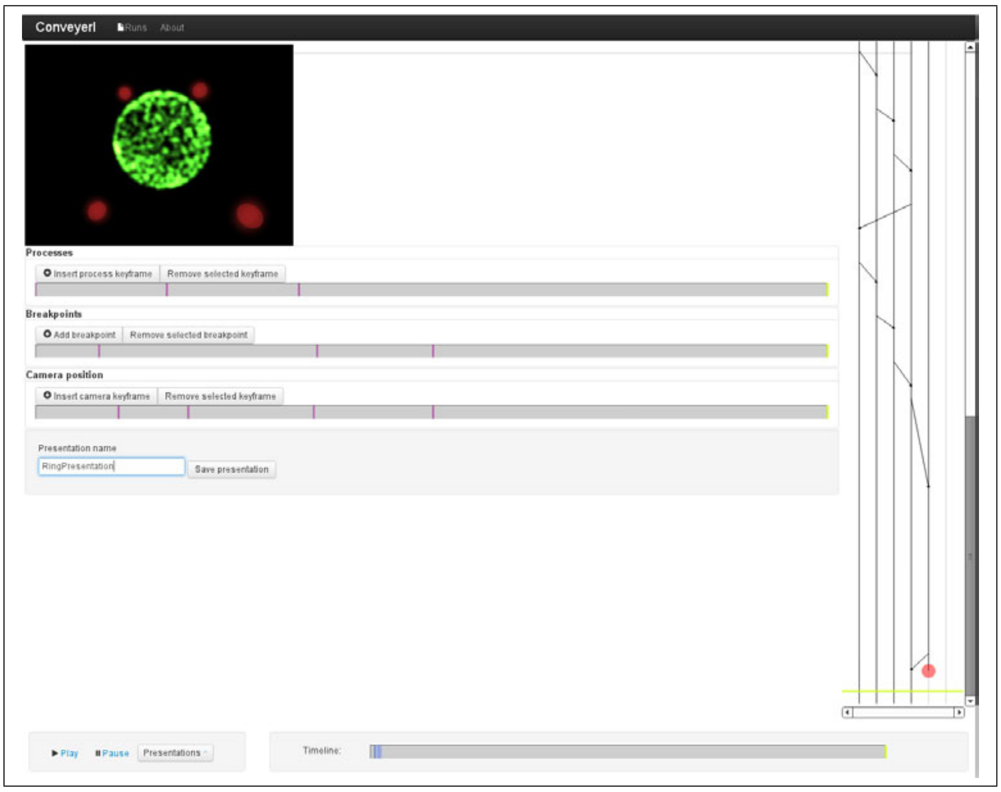
The controls have a play and stop button, a time line and a presentation selection menu. The play and stop buttons control the animated visualisation. The timeline displays the uncompressed real time of the recorded events and can also be clicked to jump to different times. The presentation selection menu can be used to load previously saved presentations.

4.4.5 The presentation editor

By using the presentation editor, the user can define presentations which can be saved to the back-end for later use. Figure 5 shows the editor view.

Positions of processes and the camera can be defined by using key frames. The user can reposition processes by dragging them into position with the mouse, and then record where they are at specific times. The animation framework will interpolate the positions smoothly at times between key frames.

The camera position and rotation can also be recorded, which allows a user to zoom in at different processes, or rotate the camera for a visual



Presentation name: RingPresentation

Figure 5: The editor view, used for creating presentations from a visualisation. It is similar to the galaxy view, but the main visualisation window is smaller and the interface is dominated by the presentation editor controls and timelines.

effect.

An important feature of presentations is breakpoints. They allow a user to define specific times at which a playing animation will pause and wait for the user to continue playing. This works similar to how a slide show is used with transitions between slides. It allows a lecturer to key the visualisation to a presentation.

The editor is based on three different time lines for the properties which can be controlled: process positions, breakpoints and camera. The time lines are synchronised to the current time of the visualisation which means the user can use the event graph to navigate to a time (eg. a message being sent) and insert key frames for that time.

To insert a key frame the user first goes to the desired time, arranges the visualisation as desired and manually saves a key frame for the state by pressing the corresponding "Insert key frame" button. Key frames are added as vertical bars to the corresponding time line. The bars can be selected and moved to tweak the animation. They can also be removed.

To save the presentation, the user names it and presses the “Save presentation” button. This will save the presentation to the back-end. Multiple presentations can be defined for every run, and does not alter the sampled data for the run.

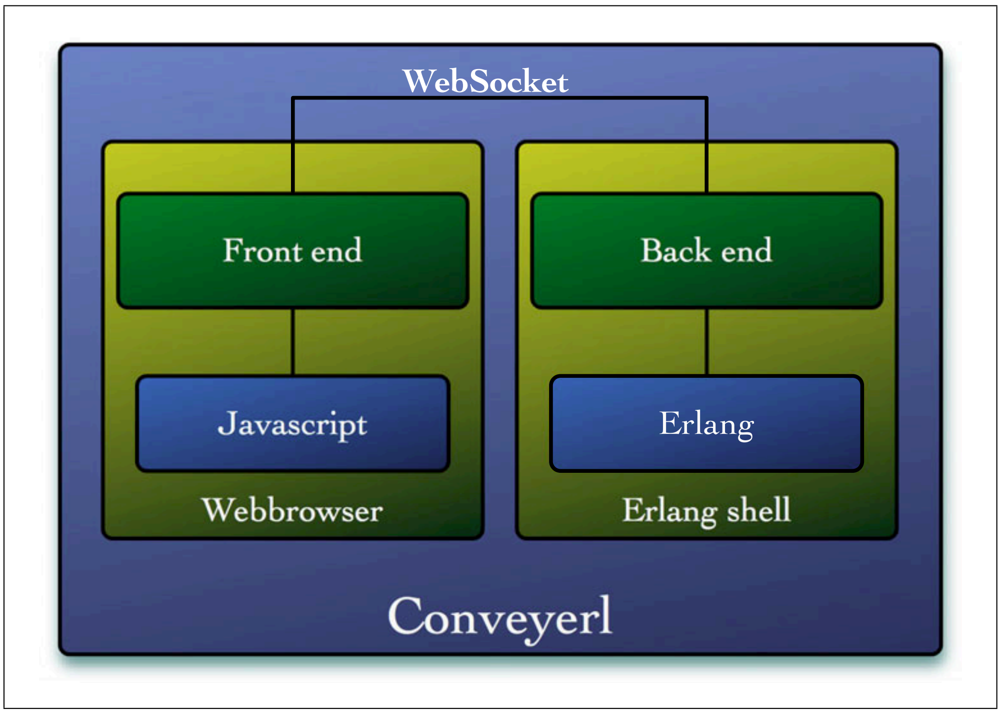


Figure 6: Conveyerl architectural overview.

5 Implementation of Conveyerl

In this section, we explain the different parts of Conveyerl in detail. The section starts by giving an overview of the architecture of the back-end and front-end, and continues with a discussion of some of the technical details.

5.1 Architecture

Conveyerl is divided into a back-end running in Erlang and a front-end running in a web browser. See figure 6. The back-end is used to sample programs, analyse samples and serve them to the front-end. The front-end runs in a web browser. It is made up of HTML which is generated by the back-end coupled with JavaScripts which runs in the browser.

5.2 Back-end

The back-end is divided into separate parts, see figure 7. It has a user interface module through which the program is used. Below is a description of the back-end modules:

conveyerl This is the module the user interacts with. It wraps the other modules in a single, easy to use interface. It's the main process and controls the other parts of the program.

runs Takes care of abstracting away the file system interaction and the organisation of *runs* - a run is one particular sampling of a given program. The runs module organises all the data for the run: the trace data, the analysed trace data and presentations saved for the run.

sampler Handles starting, tracing and stopping of the program the user wants to analyse. The sampler has a filter stack used to transform and filter the trace messages received. When the sampling stops, the sampler saves the trace data to secondary storage with the help of the runs module.

data_server The data server provides the web server with data. It runs the analysing filters on the trace data to build the data used for the visualisation. It has a clearly defined filter stack which is easily extended with new filters.

data_wrapper Acts as a wrapper for the data used by the `data_server`. It has interface functions to manipulate the `data_wrapper` record and functions for serialising the data to JSON.

web_server Is the interface to the front-end. It uses the embedded Erlang web server `misultin` to serve HTTP requests to the front-end. JSON is used to encode the analysed run-data from the back-end to the front-end. The web server uses a third party library called `erlydtl` to generate the HTML for the front-end user interface. A WebSocket is used to store and load presentations for the front-end.

5.3 Communication between front-end and back-end

Our decision to use a web browser as the platform for the front-end presented us with different ways of handling communication between the front-end and back-end.

Since the system we were building should be interactive we did not want to reload the whole web page when the user requested data from the server. The data for a run is sent to the client with the initial response when a visualisation is shown.

Early on we used WebSockets to be able to continuously push data to the client. One important factor in the decision was live visualisation. Using WebSockets at that stage meant that parts of the architecture could remain unchanged if we wanted to introduce live visualisation in the future.

Pushing all data over a WebSocket created difficulties since all components had to wait until the data was available until they could be initialised.

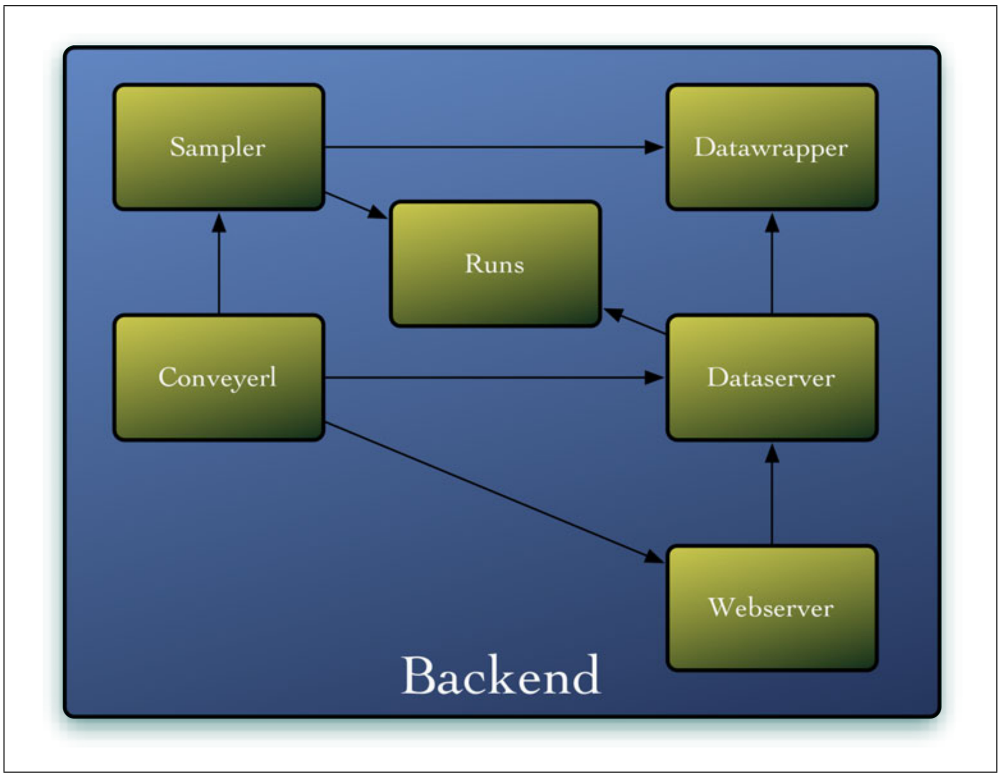


Figure 7: Back-end architectural overview.

To remove this problem we embedded the run-data in the HTML response sent to the client.

WebSockets are used for storing user data, such as presentations. These can be stored and loaded without having to reload the page. It also allowed a user to create a presentation in one place and access it from a different browser at a later time.

5.3.1 Data format

For the inherently different back-end and front-end to communicate with each other, we had to choose a data serialisation format. In the back-end, all data were Erlang terms. We wanted this encoding to add as little complexity as as possible. We considered BERT and JSON.

We chose to use JSON to encode the data between the front-end and back-end. A major reason for the choice was that JSON translates almost directly to JavaScript objects, which meant the JavaScript programs could be less complicated. We used BERT to encode message payloads since it was important to know exactly which Erlang terms were used when processes communicate.

The back-end uses `rfc4627` to encode the analysed data of the run [22].

We wrapped `rfc4627` in our own module to make it easier to replace in the future. Message payloads are encoded using Erlang's `term_to_binary/1`, followed by a base 64 [1] encoding so it can be sent together with the JSON data. The front-end uses BERT-JS [23] to decode these message payloads.

5.4 Front-end

The front-end is written in JavaScript. We had to make decisions on how to structure the code and chose to mimic an class-based organisation with special functions used to create new instances of objects with the required properties. We enforced a strict separation of different parts of the front-end. The parts are loosely coupled together with a publish/subscribe system.

JavaScript libraries

We used a number of third party libraries:

jQuery We use jQuery mainly to interface with DOM elements and event handling. At first we tried to do all handling of the DOM on our own but we ran into cross browser issues and decided to use jQuery.

raphaël Is used for basic graphical elements such as the time navigation widget and event graph.

Three.js A library to simplify working with webGL. We use it in the central visualisation component.

underscore.js Utility library which we use to simplify general JavaScript programming.

BERT-JS A JavaScript library to parse Erlang's external format. We use it to be able to visualise the data structure of Erlang messages.

Twitter Bootstrap We use the JavaScript components of the bootstrap toolkit for the user interface.

5.4.1 Front-end components

We divided the front-end into several modules, see figure 8. The main components of the front-end were:

conveyerl core The core component is responsible for setting up a WebSocket to the back-end and control the analysed data, which the other components can request. One important component of the core is the model object which has functions for normalising time.

[1] Base 64 is a way of encoding binary data using ASCII characters. This makes it possible to send binary data over a text-only protocol.

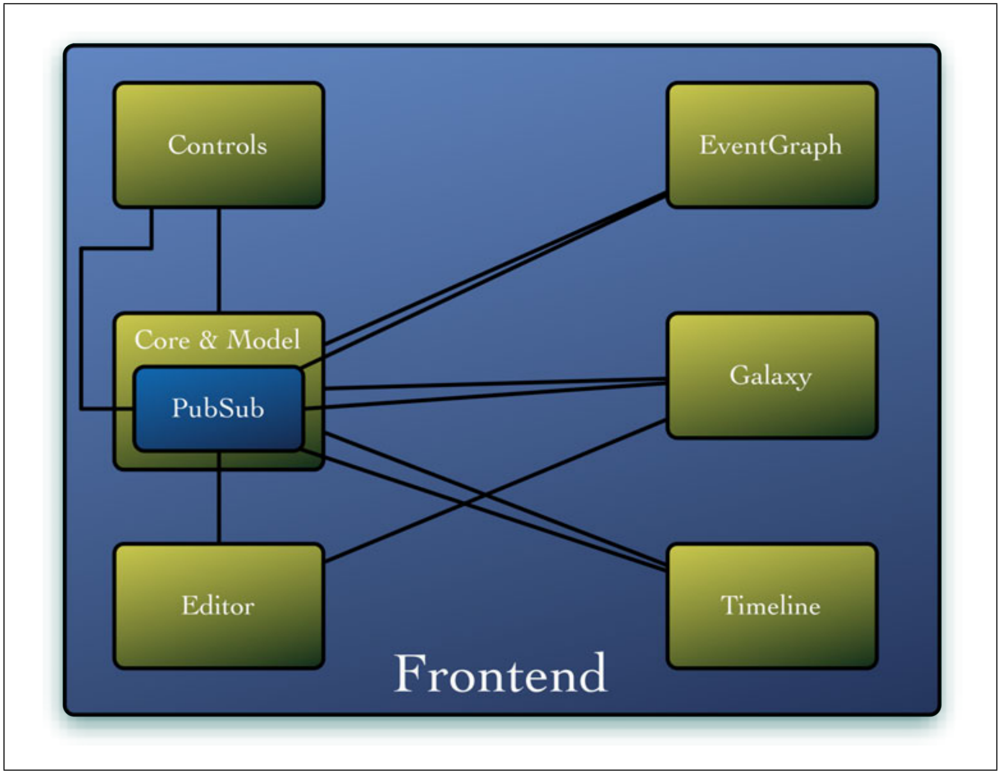


Figure 8: Front-end architectural overview.

Animations A general purpose key-frame based animation framework which allows any attribute of any object to be animated. The framework is small, flexible and powerful.

Subscription service A central service which other components can define channels to subscribe to. The components can publish arbitrary events to the channels, which all subscribers will receive.

Galaxy view The galaxy view is the main visualisation component. It uses 3D graphics to visualise the programs execution. The name comes from the fact that it uses imagery from astronomy to visualise a program. Processes are displayed as star-like objects and messages are sent as capsules.

Event graph The event graph plots events between processes over time. It shows spawn events, exit events and messages.

Timeline The timeline is a control widget which can be used to navigate the time of the visualisation.

Player A small widget which can be used to control the play-back of the visualisation as a movie.

Presentation Tools to define a presentation. The user can insert key frames for the processes positions in the galaxy view. It also gives the user the ability to define breakpoints in the visualisation so the visualisation can be incremented between specific chapters, which a lecturer can use to key his presentation.

5.4.2 Conveyerl core

The core handles the communication with the back-end. It's responsible for parsing and controlling the data of the visualisation. It has a model object which encapsulate the data of the run received from the back-end.

Other components can request the data from the model in different formats. An important feature is the ability to get the events of the run in a time-normalised format. This means that interesting events are expanded in time, while uninteresting (such as the huge time spans when nothing happens in a program) are compressed.

5.4.3 Animations

Our animations framework consists of two separate classes `AnimationData` and `AnimationChannel`. AnimationData is the main class that a user of the framework will be using. AnimationData works on objects and lets you define channels on either attributes or on functions.

If key-frame **K1** exists at time 1 with value 1 and key-frame **K2** exists at time 2 with value 2. The framework can interpolate values for the channel for all times between 1 and 2. Currently it supports two different interpolation types — or "tweens" as they are typically called — boolean and linear. The boolean type has a discrete behaviour and only changes the values on the specified key-frames. The linear type creates a linear interpolation between the values of the last and the next key-frame.

This lets us define an animation that changes the value of attribute **x** on object **A** first from 10 to 20 and then from 20 to 10. At the same time the property **y** on **A** can change from 0.4 to 1. The attributes **x** and **y** would both have a separate channel.

Defining an animation is really easy: first create a AnimationData object, then create the channels that you need on that object, and fill the channels with key-frames:

```
var animationData = new CONVEYERL.AnimationData(object);
animationData.addAttributeChannel('attr1');
animationData.addAttributeChannel('attr2');

animationData.insertKeyFrame('attr1',  0, 0);
animationData.insertKeyFrame('attr1', 10, 1);
animationData.insertKeyFrame('attr1', 25, 2);
```

For the animation to play you need to repeatedly call `AnimationData.update` with the current time.

```
  function update(time) {
    animationData.update(time);
}
```

5.4.4 Subscription service

The subscription service is used to transmit messages among the different components. In order for a component to subscribe to a particular type of message all messages are transmitted over *channels*. A channel is defined and identified by a string for instance “time” or “current_frame”.

When you — as an object — want to subscribe to a channel you do so by calling the `SubscriptionService.subscribe` method with the channel name as the first parameter, your self as the second parameter, and the callback function as the last parameter. The reason that you must pass yourself along is so that in the callback function `this` still points to `this`. Passing yourself along also lets us avoid cycles when you are publishing to a channel that you also are subscribing to. In order to break these cycles you can publish messages and *specifically* say that you don’t want those messages published to your self.

Internally the subscription service contains one object which serves as a dictionary where channels are bound to their names. A channel in that dictionary is simply an array with callback objects that contain the callback function and the object that function is called on.

When you — the object — want to publish a message you do so by calling the `SubscriptionService.publish` method with the channel name as first parameter and the message as the second parameter. If you don’t want to publish the message to yourself you pass yourself as third parameter and true as the fourth parameter.

Internally the subscription service loops over the channel’s array and then calls each callback function.

5.4.5 Galaxy view

The galaxy view is a 3D graphics component which takes the analysed data and builds a 3D world from it.

It has its own representation of the data, where processes and messages are assigned to 3D objects. The view can be interacted with by moving processes and moving the camera.

The visualisation uses an automatic layout. It’s based on a force based graph drawing algorithm to find good positions for the processes. The measure used to determine how strongly processes should be connected to each others is the amount of messages they send to each other.

5.4.6 Presentation mode

The Galaxy view has a presentation mode in which the visualisation can be controlled by using key frames. The user can define key frames for the position of processes and the position of the camera. The user can also define breakpoints, points on the timeline where the visualisation will pause and wait for input before continuing. This can be used to design presentations.

5.4.7 Event graph

The event graph displays the processes as separate lines over time. Messages are shown as arrows between lines over time. It uses raphaël to create SVG graphics.

5.4.8 Timeline

The timeline is a navigational tool which displays the complete time of the run and shows at what times events occur. The currently selected time is shown as a marker and the user can jump to different times by clicking the timeline.

5.5 Tracing Erlang programs

To be able to follow the execution of a program we used Erlang's built in trace-functions described in section 2.1.3. The trace functions worked very well for getting most of the information we wanted but there were some properties which we were not tracable:

Process state The tracer gave us context switching, but could not give us information of when a process was waiting in a receive statement nor could it tell us which expression in the source code a process was executing.

Mailbox We couldn't get the contents of the mailbox by using the tracing and could not get informed of when a process retrieved a message from the mailbox.

Memory usage The trace messages gave us garbage collection messages, which in turn gave the size of the process heap and stack, but we would have to rely on the garbage collector to actually run, which would not necessarily reflect the actual memory use of a process.

Monitors The trace functions clearly show spawn and link events, but not monitor events.

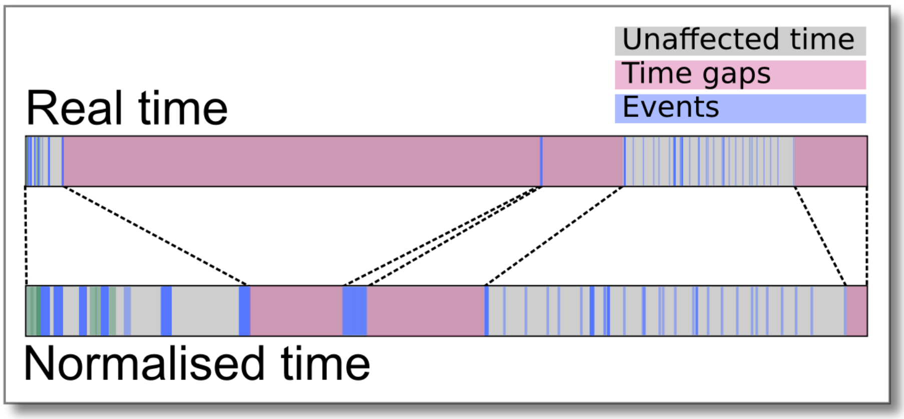


Figure 9: Illustration of the time normalisation.

To get this information, we periodically poll the traced processes by using the `erlang:process_info/2` function. This gave us the information the `erlang:trace/3`-function didn't.

One problem with the tracing of function calls was that it made the trace data explode in size. A source of this was the use of function calls in list comprehensions. If a program made heavy list processing, a function would get called for every element in a large list, which in turn made the number of trace-messages for calls increase dramatically.

We opted to using polling instead, where we simply check where a function is at a certain time. This way we can reduce the amount of information at the cost of exact knowledge of what functions a process calls. If the user knows that the number of function calls in a program is manageable, the user can easily switch on the call tracing.

5.6 Normalised and Real Time

Since the goal was to visualise events such as messages being sent and processes spawning or exiting, we discovered that a lot of time the program was idle. Even relatively short periods of idleness proved to be long in comparison with the time it takes for a message to be sent.

On a local node, sending messages between processes is in the order of a couple of microseconds. This made it impossible to visualise events of the system in real time. When we scaled up the time linearly, the idle periods became unreasonably large. If the time was scaled up so that the time it took to send a message were a couple of seconds, an idle time in the sample of 1 millisecond would take a quarter of an hour of scaled time to play.

We realised that the real view of time was largely useless for visualising the program since the short time-span of interesting events made it difficult to distinguish between them in non normalised time.

We solved this problem by scaling time periods differently depending on what occurred. Long periods of idleness were scaled down and periods when interesting events occurred were scaled up in time. Figure 9 illustrates the concept.

6 Evaluation

In this section we evaluate the software we created in relation to the design criteria we defined. We also describe some of the decisions we made in while creating the visualisation tool, such as how to handle time and animations.

6.1 Student use case

For the student use case refer to section 3.0.3 for our requirements.

6.1.1 Interactivity

An important design criteria was that the tool should be interactive, and we concluded that being able to control the time of the animation was one of the most important interactive features. Playing, pausing and jumping in time was required to be able to really get information about a program from the visualisation.

The user is able to reposition processes, but we believe this to be less important from a learning point of view, and we used it mostly when the processes were badly laid out by the automatic layout. Being able to control the camera doesn't convey information about the program and mostly gives the user a sense of the space which the processes occupy.

6.1.2 Installation

To simplify installation, we used web browser for the front-end. Compatible browsers are readily available for many different operating systems and many users might already have one installed. We managed to keep the installation relatively simple. Apart from Erlang and a web browser which supports WebGL (see section 2.7, the user needs basho's `rebar` build script which is used by many Erlang projects [4]. Conveyerl also has some dependencies on third party Erlang libraries, but these are installed transparently by the `rebar` build process.

6.1.3 Automatic visualisation

After a sample has been created, the creation of the visualisation is fully automatic, and the user doesn't need to specify any special parameters to get a visualisation. The automatic layout can give bad positions which could be confusing. These can be rearranged by a user, but the automatic layout should be improved.

6.1.4 Beauty

The main visualisation uses 3D-graphics. We also considered using 2D vector graphics for the process visualisation. We chose 3D graphics since it allows a lot more of flexibility when it comes to producing visual effects.

Beauty played a minor role in the development efforts. Some of the visual effects (like the explosions when processes exits) can be captivating, but not much effort has been put into tuning them for aesthetic effects. An important question to evaluate in the future is what impact special effects would have on the learning process, if and when they could be beneficial or detrimental.

6.2 Teacher use case

For the teacher use case refer to section 3.0.4.

The main feature for the teacher use case was the ability to create presentations based on visualisations. As a presentation tool for the teacher we made some headway but still have some way to go. Since we created the presentation tool as an extension to the visualisation for the student use case we did not have time to implement all the design criterias.

6.2.1 How to create presentations

When deciding how a user should create a presentation we saw two distinct possibilities:

Source code annotations The lecturer could write programs annotated with commands to control the visualisation. This would decouple the lecturer from the visualisation tool actually used in the end.

What You See Is What You Get (WYSIWYG) With WYSIWYG, the teacher would define the presentation inside the visualisation by moving objects into position, manipulating the camera and add key frames for their values at different times.

We choose to implement a WYSIWYG solution. We saw a clear advantage with being able to tune the visualisation post mortem, since a lot of times it might not be obvious how the program you write should look and be presented. Another important reason was that we extended the visualisation for the student use case. Going from a working visualisation to a presentation editor tool was not a long step, since most of the framework was already in place. We also believed that designing a language would be outside the time scope of the project.

6.2.2 Editor

The editor for the presentation tool has a major usability drawback. The time lines which are used to insert, reposition and remove presentation key frames are based on real time. When creating a presentation, key frames are often inserted at times with a lot of events, and these time-spans will often be very short in real time. The timelines would be more useful if they used the normalised time like the event graph.

6.2.3 Implemented design goals

The following is a list of the functionality we have implemented and some notes on their implementation:

Positioning processes The processes positions can be changed at different times, and the positions can be recorded as key frames. Processes positions will smoothly interpolate between key frames.

Positioning the view The camera rotation and position can be recorded as key frames. This allows the teacher to zoom in on different processes at different times.

Breakpoints The teacher can define breakpoints which will pause the visualisation when it is played, and wait for the presenter to hit the play button to continue playing.

Store and load presentations Presentations can be saved to the back-end and be loaded at a later time. The presentations does not alter the sampled and analysed data, and multiple presentations can be recorded for the same run.

6.2.4 Design goals not implemented

Due to time constraints, a number of criteria for the presentation tool were not implemented. Below is a list of the criteria which were not implemented, and a short discussion about them:

Source code display This is a feature which we believe would be really useful. It would allow the visualisation to work like an instructional movie and could be used for students to learn Erlang on their own.

Annotations We believe this feature is of lower importance. A major issue with implementing it is how to do it well from a user interface point of view.

Distribute visualisations This is a feature we believe could be useful. Distributing visualisation as static HTML/JavaScript files would be a great way for a teacher to give teaching material to students. It could completely remove Erlang as a dependency, and only require a WebGL compatible web browser.

6.3 Visualised properties

Processes are central to our visualisation, and their visualisation has been a important area of development. Below is a description of the properties we visualise, how they are visualised and a short discussion of the visualisation:

Process identity We assign processes their colour based on the function that the process was spawned with. Processes spawned with the same function have the same colour. This worked out well as way to differentiate processes from each other. Although the success may be subject to the program that is visualised and may not work as good on all programs. A case where this becomes less useful is when processes are spawned with a call to `apply/3` since it can call arbitrary functions. We saw this is some of our test programs. The identity can be discerned by highlighting the process with the tool tip which gives the arguments to `apply/3`. A solution could be to add manual exceptions to the algorithm which assigns colors, to handle `apply/3`.

Process state The process running state is based on whether it's context switched in or out. It is displayed as a change in the brightness of that process's colour. This gives a good overview of which processes are running and which are idle. The visualisation gives a clear idea of when processes execute, even on programs with about 50 processes.

Life ranges Life ranges are clearly displayed both in the event graph and in the galaxy view. In the galaxy view processes are only visible as long as they are alive and both the spawn and exit events are animated. The spawn events are shown as a cloud of matter going from the spawner to the place of the spawned process and the exit events are shown as a colourful explosion. The spawn effect is useful for getting actual information about which process spawns which, but the exit effect is mostly a visual effect.

Messages Messages that are sent between processes are displayed as projectiles. When the user hovers the mouse pointer over one of the messages, the message content is displayed as a tool tip. During the development of Conveyerl we tested the tool with programs which sends relatively few messages, for which it works well. For programs where huge amounts of messages are sent frequently, using abstractions for the messages would probably be required.

Relationships We only visualise links between processes. They are visualised as lines, using the color of the two processes. This gives a quick overview of process link dependencies. The links does not influence the automatic graph layout, which can create confusing layouts.

If we visualised more relationships than links, a more informative visualisation would be needed to differentiate between them. As it is now, the visualisations for the links doesn't contain strong semantic properties, and could be mistaken for monitors or communication relationships.

6.4 Usability

The focus of this bachelor thesis has been on creating the software foundations for the visualisation. The usability has not been properly designed and evaluated. Some examples of this is the layout of the visualisation controls and the time lines in the presentation editor. We believe there is room for improvement from a usability standpoint.

6.5 Normalised and Real Time

How time is handled proved to be an important factor in the visualisation. We started by using a linear scaling, where all time was expanded so that short-lived events would be visible. This led to event less periods being far too long. We implemented a scaling of time, which compresses idle time periods and expands eventful ones.

The time scaling led to a more informative visualisation. The major drawback is that the visualisation becomes a false representation of time. The events are still in the correct order, and the compression/expansion does not violate causality. If the real time it takes for events is important when using the visualisation, the scaling have to be turned off.

We used compressed time when playing the animation and creating the event graph, but used real time for the main time navigation control which we call the timeline. The timeline is useful to give a sense of the real relation of events in time, but to understand the interaction of events the normalised time is better. Trying to navigate to a time on the timeline when a message is being sent is extremely hard, since it's only a minuscule fraction of the entire time. During development we found that we used the event graph almost exclusively for navigating time.

While the normalised time present a false description of time, it's more useful when it comes to conveying meaning about the properties of programs we wanted to visualise, especially messages and processes. We believe that any visualisation which wants to visualise ephemeral properties of computer programs will have to do something similar to normalise time.

6.6 Live visualisation vs. post-mortem

We had to decide whether to make the visualisation use data from a live running system, or post-mortem. Live visualisation is when the visualisation is created directly from a running system. Post-mortem visualisation is when the visualisation is created from a recording of a program. Both types had their advantages and disadvantages.

6.6.1 Live visualisation

Advantages

- Live visualisation gives us the opportunity to interact with a running system and see the results of the actions. It could be very useful for trying to provoke errors. It could also give students immediate feedback on their actions.
- While live tracing, only a subset of all the traces has to be saved. This could allow for much larger systems to be visualised for a longer time (might even be indefinite).
- With live tracing, it would be possible to plug into a running system without having to set up the system beforehand. (Could probably be done with post mortem as well, just gather a trace and analyse it off line).

Disadvantages

- Live tracing a program could become complicated, especially when causality is important. Deciding when you've gathered all the messages for a certain chain of events can be difficult.
- With a live visualisation, it might be more difficult to decide which trace messages to use and see global trends in the data.
- A live visualisation might be non-reproducible. This could be solved by saving a log of a live running system.
- A live visualisation will have difficulties in regards to time. Since interesting events can be very short-lived, it is difficult to visualise them properly and still keep the live nature of the system.

6.6.2 Post-mortem visualisation

Advantages

- Much easier to gather the data and decide on what events are causal.
- Simplifies experimenting with the data, testing different ways of processing it or visualising it.
- Simplifies analysis, especially if different trace messages have dependencies.

- Easier to visualise “global” behaviour of a system, properties spanning the whole execution.
- A trace, once gathered, would always give the same visualisation, like a recording. Useful for analysing the behaviour of the system. Also important for an educator or public speaker who wants to prepare animations beforehand.
- A post-mortem visualisation modify the time of the run. Short-lived events can be expanded and uneventful periods can be shortened to focus on visualising the interesting parts of a running program.

Disadvantages

- With post-mortem visualisation, it would be impossible to interact with the visualisation.
- All the trace data has to be saved to disk. For large systems this could be unfeasible.

We choose to work with post-mortem visualisation. During the development phase this allowed us to gather a sample once. We could then work with the same trace through the evolution of the implementation. It also simplified the work on the front-end. By being completely aware of everything that was going to happen, the front-end could pre-calculate positions of processes and messages.

6.7 Animation framework

We created our own framework to handle animations, one of it’s strengths is the ability to change any attribute on an arbitrary JavaScript object. The framework defines animation channels, where every channel is tied to a specific attribute of an object. Key frames are defined for the channel to change the attribute value at different times. For times in between key frames, an interpolated value is calculated. See section 5.4.3 in the implementation section for more details.

In our animations we needed to be able to pause animations at any time, as well as skip to a specific time, and get the correct animated attributes for that time.

A lot of the existing JavaScript animation frameworks use chains of callback functions for animations. Once started they continue to play until done. The only way to pause them is to stop the animation completely, which means you can’t pause and later continue playing from where you were.

Another limitation is that many of the frameworks only animates DOM objects. In our case, most of the animations did not use the DOM at all.

Since our software was written in JavaScript, we wanted a framework which could animate any arbitrary JavaScript value, regardless of it’s context.

7 Related Work

7.1 Erlang trace analysis

Previous Erlang trace analysis has been carried out by Thomas Arts and Lars-Åke Fredlund [3]. They used Erlangs trace functions to analyse programs based on the generic OTP behaviours, `gen_server` and `gen_fsm`. Their tool creates a state graph which can be visualised with tools like graphviz and daVinci.

7.2 Software Visualisation in Education

Hundhauses et. al. performed a meta study of algorithm visualisation in education [14]. They studied the effectiveness of AV performed in 24 different experimental studies. They conclude that how students use a visualisation technology is what matters. Visualisation can contribute to learning, but the form of the learning activities more important than the form of the visualisation.

Urquiza-Fuentes and Velázquez-Iturbide has performed a survey on the success of software visualisation in education [24]. In their survey they concentrated on studies which had positive results when using software visualisation in education. They offer recommendations for different engagement levels of software visualisation.

Bennedsen and Schulte performed an experiment using the BlueJ visual debugger for Java programs [6]. They evaluated the effect the visual debugger had on how well students learn object interaction. They found that the visual debugger had a positive effect on understanding, but not that students using it performed statistically better than those manually tracing program execution.

7.3 Process Visualisation

The kind of process visualisation we have experimented with is close to what's used by Gustavo et. al. to visualise distributed systems [13]. In their case, they use the visualisation tool OverView. They present two different layouts, a force based and a hierarchical. Especially the force based shares similarities with how Conveyerl visualises processes.

Björn-Egil Dahlberg has created visualisation of Erlang processes [8] which is similar to Conveyerl. He uses Erlang and wxErlang to visualise the processes of another Erlang node. The visualisation he creates is based on live tracing and uses a force based graph layout similar to ours.

Kresten Krab Thorup created a software visualisation similar Conveyerl called `erlubi_tracer` [15]. He created a live visualisation of processes running on a node, similar to fgraph. He created an Erlang module to interface with the Ubigraph graph library. An example of the usage of the tool was connecting the graph library to Erlangs trace tool, which gives a 3D graph of all the running processes on an Erlang node.

Both `fgraph` and `erlubi_tracer` focus more on the structure between processes. Since they both depend on live tracing, messages are not clearly visualised. Time is a fundamental property where Conveyerl differs from these visualisations, since it normalises time to visualise properties which would otherwise be too short lived to be observable.

8 Future work

8.1 Other Properties to visualise

The properties of Erlang programs which Conveyerl visualises are limited. In the future we would like to be able to visualise the following additional properties:

Process state We would like to be able to show where (in the source code) a process is currently running. This could be done with differing level of coarseness. Being able to see exactly what function a process is currently in is possible with Erlang's trace tools, but increases the sample data enormously (every call and return is traced). This would probably have to be combined with trace patterns supplied by the user, which was out of the scope for our use cases.

The processes could also be probed to get the actual line of code it's currently at, which is already done at the back-end, but not visualised in the front-end.

Life ranges We would like to visualise the exit reason for processes. This could either be as text or as different animations for the exit event.

Relationships We would like to visualise monitor relationships between processes, but tracing this isn't readily available by the tracing functionality.

Resource usage We would like to visualise the memory usage of processes. All the support is already available in the back-end and the sampled data, but has not been implemented in the front-end.

Mailbox We would like to visualise how many messages are waiting in the process mailbox. We would also like the user to be able to inspect the contents of the messages.

Communication patterns We would like to show trends in communication between processes. For example, the volume of messages sent between processes over time and how it changes. This would be useful if a system sends a huge amount of messages, and the contents of individual messages isn't important, but which processes communicates with which is.

Data structures We would like to be able to visualise data structures, allowing a user to explore them visually.

8.2 Other use cases

We envisioned two other use cases for this software: development and art. These are use cases we never had time to look into due to time constraints.

8.2.1 Development

To create a visualisation useful in development we would need to do a lot more research. We believe that development places higher demands on exactly what is visualised, how it is visualised and how the visualisation fits with the work flow of a developer. The visualisation could be used as a visual debugger to quickly diagnose problems in a Erlang program.

Depending on the scale of the system, a developer might need different types of information. We distinguish between two perspectives for development:

Micro This perspective is focused on small systems or smaller parts of a bigger system. It would be the scale that most current debuggers covers. Things like CPU time, message queues and so on. The level of detail would be high, and most interesting properties of individual processes might be presented. An important question is whether a visualisation of this level would actually provide the developer with anything useful, or if the current text based debuggers cover this use case better.

Macro In this perspective the developer analyses large systems with thousands or millions of processes. It is difficult to predict what types of patterns that might emerge. Therefore it would be important for developers to easily modify the program so that it visualises the properties that the developer is interested in.

An important aspect of large Erlang programs is that they're seldom implemented only in Erlang. Getting information about where the program spends time (even in Erlang-external parts) could be important.

Another issue is how to model the system and the visualisation. What would an analogy be for how a huge number of processes interact. Would it behave similar to physical phenomenons like gasses, brownian motion etc?

8.2.2 Art

Art was one of the starting points for this project. The inspiration came from a desire to to see the complex systems programmers have created. Just by trying to imagine the structure of a complex software system, we feel that the structure can be truly awe inspiring.

This use case would be about giving programmers the ability to transform their programs into works of art, visually echoing the structure of the software. To do this, we would like the visualisation to be easily tweaked and tuned. Aspect such as colour, graphical primitives and visual effect should be easy to add and control.

We see the users of this kind of visualisation as being programmers with an philosophical interest in computer programs. Programmers who wishes to examine the questions: "what *is* a computer program?" and "what does a computer program look like?". Programmers such as ourselves.

8.2.3 Going further

The next step is to refine the usability of the tool, by evaluating its usefulness in education, working together with teachers who have experience in teaching Erlang. For this step, we would need to conduct a more thorough investigation into the studies done on the effectiveness of software visualisation in education.

Extending the capabilities of the presentation tool would also be an important next step as discussed in the evaluation. We believe that giving teachers the ability to easily couple visualisation to source code would be an important addition. Another feature we would like to implement is the ability to export a visualisation to static HTML and JavaScript files, bundling a visualisation in a portable, easy to install and use package. The only requirement would be a web browser supporting WebGL and HTML5.

Scalability is also a desirable future goal. At it's current state, the usefulness of the system is limited to about 50 processes. Experimenting with visualising huge distributed systems would require sophisticated algorithms for the layout and grouping of processes. Certain abstractions would likely also be needed in order to get a grasp of the run. It would also require techniques for storing huge amounts of data, since the system uses post-mortem analysis.

9 Conclusions

The main outcome of our thesis project has been a software for analysing and visualising Erlang programs in a browser. The focus of the project has been on getting a basic framework in place. During the development, we have had education in mind. The software created has a modular design which is easily extended.

We have not evaluated the use of the visualisation tool in education and can not draw any conclusion about its usefulness for new Erlang programmers. We have laid the foundations for a tool which students can use in order to learn Erlang but we believe it still have some way to go before being used in class rooms.

In this report we have presented the development of the Conveyerl visualisation tool for visualising Erlang programs. Conveyerl uses post-mortem analysis of Erlang trace data to create 3D visualisations in web browser. While developing Conveyerl we have focused on features relevant for students and teachers. The report has given an overview of the tool and concentrates on the architecture and implementation details of the software. This bachelor thesis project has laid the foundation for the software visualisation tool. We have not evaluated the usage of Conveyerl in education.

10 Acknowledgements

We would like to thank Joe Armstrong without who's enthusiasm and support this project would never have started.

Our thanks also goes out to Christian Schulte who continues to deliver valuable insights and for helping us navigate the administrational maze of KTH.

References

[1] Christopher Alexander, Sara Ishikawa, Murray Silverstein, *A Pattern Language: Towns, Buildings, Construction* , Oxford University Press, USA, 1977.

[2] Joe Armstrong, *Programming Erlang: Software for a concurrent world*, The Pragmatic Bookshelf, Raleigh, N.C. 2007.

[3] Thomas Arts, Lars-Åke Fredlund, *Trace Analysis of Erlang Programs*, ACM SIGPLAN Erlang Workshop '02, Pittsburg, PA USA, 2002.

[4] basho, *rebar — A sophisticated build-tool for Erlang projects that follows OTP principles*, Retrieved June 25, 2012, <`https://github.com/basho/rebar`>.

[5] Martin Beck, Jürgen Döllner, *Towards Automated Analysis and Visualisation of Distributed Software Systems* SOFTVIS'10, October 25–26, ACM, Salt Lake City, Utah, USA, 2010.

[6] Jens Bennedsen, Carsten Schulte, *BlueJ Visual Debugger for Learning the Execution of Object-Oriented Programs*, ACM Transactions on Computing Education, Volume 10 Issue 2, June 2010, Article No. 8 ACM New York, NY, USA, 2010.

[7] Douglas Crockford, *The application/json Media Type for JavaScript Object Notation (JSON)*, July 2006 Retrieved June 25 2012, <`http://tools.ietf.org/html/rfc4627`>

[8] Björn-Egil Dahlberg, *fgraph — Physics engine for graph drawing written in erlang for use in wxErlang or standalone*, Retrieved May 25 2012, <`https://github.com/psyeugenic/fgraph`>.

[9] Ericsson, *Erlang Reference Manual: External Term Format*, Retrieved May 25 2012, `http://www.erlang.org/doc/apps/erts/erl_ext_dist.html`.

[10] Ericsson, *Erlang Reference Manual: Data Types*, Retrieved May 25 2012, <`http://www.erlang.org/doc/reference_manual/data_types.html`>.

[11] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Boston, USA, 1995.

[12] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, Ioannis G Tollis, *Algorithms for drawing graphs: an annotated bibliography*, Computational Geometry, Volume 4, Issue 5, Pages 235-282, October 1994.

[13] Gustavo A., Guevara S., Travis Desell, Jason Laporte, Carlos A. Varela, *Modular Visualization of Distributed Systems*, CLEI Electronic Journal, 14:1-17, April 2011.

[14] Christopher D. Hundhausen, Sarah A. Douglas, John T. Stasko, *A Meta-Study of Algorithm Visualization Effectiveness*, Journal of Visual Languages & Computing Volume 13, Issue 3, Pages 259–290, June 2002.

[15] Kresten Krab Thorup, *Erlang Visualizer using Ubigraph*, Retrieved June 17 2012, <`https://github.com/krestenkrab/erlubi`>.

[16] Peter Liggesmeyer, Jens Heidrich, Jürgen Münch, Robert Kalcklösch, Henning Barthel, Dirk Zeckzer, *Visualization of Software and Systems as Support Mechanism for Integrated Software Project Control* , Proceedings of the 13th International Conference on Human-Computer Interaction. Part I: New Trends Pages 846 - 855, Springer-Verlag Berlin, Heidelberg, 2009.

[17] The Khronos Group, *WebGL Specification Version 1.0*, 10 February 2011, Retrieved June 25 2012, <`https://www.khronos.org/registry/webgl/specs/1.0/`>

[18] The Khronos Group, *Getting a WebGL Implementation*, Retrieved June 25 2012, <`http://www.khronos.org/webgl/wiki/Getting_a_WebGL_Implementation`>.

[19] Peter Lubber, Frank Greco, *HTML5 Web Sockets: A Quantum Leap in Scalability for the Web*, Retrieved June 25 2012, <`http://www.websocket.org/quantum.html`>.

[20] Clifford A. Shaffer, Matthew L. Cooper, Alexander Joel D. Alon, Monika Akbar, Michael Stewart, Sean Ponce, and Stephen H. Edwards. *Algorithm Visualization: The State of the Field* Trans. Comput. Educ. 10, 3, Article 9 (August 2010), New York, NY, USA, 2010.

[21] John Stasko (ed), John Domingue (ed), Marc H. Brown (ed), Blaine A. Price (ed), *Software Visualisation: Programming as a Multimedia Experience*, The MIT Press, © 1998. Books24x7. Web. Mar. 1, 2012.

[22] Tony Garnock-Jones, *Erlang RFC4627 (JSON) codec and JSON-RPC server implementation*, Retrieved June 25 2012, <`https://github.com/tonyg/erlang-rfc4627`>.

[23] Rusty Klophaus, *BERT-JS: Javascript implementation of BERT serialization (Binary ERlang Term)*, Retrieved June 25 2012, <`http://github.com/rustyio/BERT-JS`>

[24] Jaime Urquiza-Fuentes, J. Ángel Velázquez-Iturbide, *A Survey of Successful Evaluations of Program Visualization and Algorithmic Animation Systems*, ACM Transactions on Computing Education, Vol. 9, No.2, Article 9, New York, NY, 2009.

[25] W3C *Document Object Model (DOM)*, January 19 2005, Retrieved May 24 2012, <`http://www.w3.org/DOM`>.

[26] W3C *HTML Working Group*, Retrieved June 25 2012, <`http://www.w3.org/html/wg/`>.