Technical Report: DaViz

Hans-Dieter A. Hiep

ABSTRACT

DaViz is a software tool written in the Haskell and Java programming languages for simulating and visualizing executions of distributed algorithms. Users can define their own network structure upon which a limited set of implemented algorithms run. Users choose an order of events to determine an execution of the algorithm and rearrange the events on a timeline to construct a concurrent computation. This document reports on the software's architecture, design decisions, implementation details and lessons learned. The report concludes with future assignments for improving the software.

1. INTRODUCTION

The DaViz project consists of the design and development of a system for simulating and visualizing distributed algorithms. The goal of the project is to let students explore computations of distributed algorithms by means of an interactive desktop applications.

Tool Description.

Students can load a network description and choose from a set of predetermined suitable algorithms that will execute on the simulated network. Networks are visually represented by an interactive graph, in which nodes (processes) are connected by edges (channels). A timeline shows an interactive view of a particular execution of a selected algorithm, and students can adapt the execution by reordering events. The change is reflected in the timeline by recomputing the consequences of such a reordering on the execution of the algorithm. Furthermore, students can step through the timeline and view the changes reflected in the network. At each point in time, the network visualization updates to reflect the current state of the network. Students can select individual processes to view their internal state, and channels to view messages in transit.

In summary, the aim of the tool is to let students of a distributed algorithms course interactively explore distributed algorithms that allows for experimentation and in-depth study of the algorithms without any knowledge of to the underlying theory.

Deliverables.

The most important deliverable of the project is a software deliverable. Secondary is this technical report that serves as end-user and system maintainer documentation, describing in detail the functionality of each aspect of the system and ways of extending the system in the future.

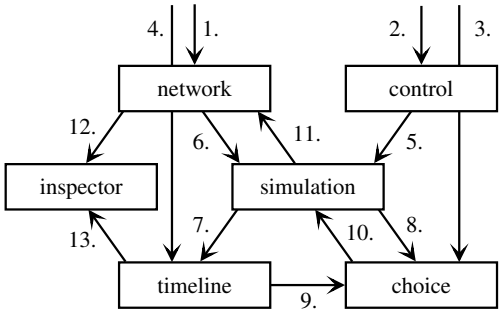
1. The software deliverable comprises a full source code repository, including the scripts for building and packaging, and a deployable software package. Additional files required for the operation of the system are included as well, including example network descriptions.
2. The technical report describes in detail the functionality and architecture of the delivered system. The contents of this report will be discussed during a demonstration at the end of the project.

Document Structure.

This report is structured as follows: in section 2, a high-level description of the chosen software architecture and a rational for design decisions is given. In section 3, the most interesting user interface features are described. This section also serves as end-user documentation. In section 4, implementation details are described. Sections 2 also contains critical reflection on the taken approach. The report concludes with section 5, listing future assignments for improving the delivered software.

2. ARCHITECTURE AND TECHNOLOGY

The system is an interactive tool which can best be described by the following information flows between system components:



Network Responsible for representing a network topology. In essence this comprises processes and communication channels.

Control Responsible for signaling other components what simulation is performed. It comprises a selection of algorithms and represents their necessary assumptions for operating.

Simulation Responsible for performing a simulation of a selected algorithm based on the user-defined network topology, and user-specified non-deterministic choices.

Timeline Responsible for representing (concurrent) events of the simulation. It comprises processes, their local events and messages between processes, graphed on a time-representing axis.

Choice Responsible for representing non-deterministic choices in the simulation. It comprises a selection of possible future worlds, depending on the notion of 'current time'.

Inspector Responsible for representing detailed fine-grained information. It comprises a table of (nested) keys and their respective values.

1. The user supplies the network component with topology information: it defines processes and the channels between processes. Additionally, the user may select any number of the two kinds of elements.
2. The user supplies the control component with a selection of the simulated algorithm, together with additional information defined per algorithm, and gives the command for starting or reseting the simulation.
3. The user supplies the choice component with a selection of the simulated future world.
4. The user supplies the timeline component with commands for changing the current time. Additionally, the user may select any number of events or messages.
5. The control component signals the simulation component to commence simulating, based on the currently selected algorithm, its additional assumptions.
6. The network component supplies the simulation component with the currently represented network topology.
7. The simulation component updates the timeline component with a trace of events and messages.
8. The simulation component updates the choice component with a trace of possible future worlds, one collection of future events per event represented by the timeline.
9. The timeline component informs the choice component of the current time, and hence the current event, to determine the selection of future events.
10. The choice component informs the simulation component of a new future world, that influences the future simulation.
11. The simulation component informs the network component of the last state per process and current messages in transit, based on the chosen future simulation and current time.
12. The network component informs the inspector component of its current selection to show detailed simulation information regarding the selected process or channel.
13. The timeline component informs the inspector component of its current selection to show detailed simulation information regarding the selected event or message. Additionally, it informs the inspector component of the current time to change the detailed simulation information of the currently selected objects.

2.1 Design Decisions

The most important design decisions in support of the former architecture are below. These decisions are not listed in any particular order.

No well-defined requirements.

The architecture is developed in accordance to the needs of the project at each stage. The network component and timeline component were identified from the start on; the inspector, control and choice components are added when requirements indicated they were needed. This *ad-hoc* style of requirements management ensured a high pace during the initial phase of development, quickly iterating the eventual architecture of the tool.

Simulation back-end.

To support the implementation of numerous algorithms, a decision was made to implement these algorithms in Haskell. The Haskell implementation is consequently integrated with the rest of the implementation by employing Frege, a project for compiling a Haskell-like language to Java sources suitable for consumption by the Java Virtual Machine. By the nature of the Frege project, there is no seamless integration between the compilation results and other Java objects. Hence we have implemented a glue layer that is tightly coupled to the compilation results of the Frege compiler, for invoking Haskell functions and extracting its computational results.

We now give a high-level outline of the data structures employed to support implementing distributed algorithms within the framework of the simulation.

We have a graph representation of the network G, where processes are vertices represented by natural numbers, i.e. $V \subseteq \mathbb{N}$, and channels are edges as pairs of numbers, i.e. $E \subseteq V \times V$, and the graph is represented as a set of edges. A channel is identified as such a pair between processes. The channel state is a list of channel messages, and a channel message is a tuple of a channel together with some message value μ. We then proceed to define a so-called process description.

Definition 1. *A* process description *is a pair of an initialization function* $\text{init} : \rho \to \sigma$ *and a step function* $\text{step} : \sigma \to R$*, where R is the set of* process results *defined as follows:*

1. *Given a receive function from a channel message* μ *to some next state* σ*, we have a* **Receive** *process result.*
2. *Given a list of possible future internal states* $[\sigma]$*, we have an* **Internal** *process result.*
3. *Given a pair of future state* σ *and a channel message* μ*, we have a* **Send** *process result.*
4. *Given a decicion* τ*, we have a* **Result** *process result.*

All is parameterized over assumption space ρ*, message space* μ*, state space* σ *and result space* τ*.*

Each distributed algorithm is described by giving a process description. For example, a tree algorithm has the following definition:

Definition 2. *The* tree algorithm *has as assumption space* ρ *a known network, i.e. a pair of a network and a single process. As message space* μ*, it has a unit value* $\cdot$*. As state space it has tuples of sets of neighboring channels and an internal state of either undef or some parent channel. As result space* τ*, it has a unit value* !*. Its initialization function is given by*

$$\text{init}(G,p) \coloneqq (\text{out}_G(p), \mathit{undefined})$$

where $\text{out}_G(v)$ *is the set of all outgoing edges in graph G for vertex v. Its step function is given by*

$$\text{step}\langle N, \mathit{undef}\rangle \coloneqq \textbf{Receive}\,\lambda(x,\cdot).(N \backslash \{x\}, \mathit{undef}) \quad (1)$$

$$\text{step}\langle \{c\}, \mathit{undef}\rangle \coloneqq \textbf{Send}\,\langle(\emptyset, \mathit{parent}\,c),(c,\cdot)\rangle \quad (2)$$

$$\text{step}\langle \emptyset, \mathit{parent}\,p\rangle \coloneqq \textbf{Receive}\,\lambda(p,\cdot).(\{x\}, \mathit{parent}\,p) \quad (3)$$

$$\text{step}\langle N, \mathit{parent}\,p\rangle \coloneqq \textbf{Result}\,!\ \mathit{if}\ |N| = 1 \quad (4)$$

Obviously, the tree algorithm has a partial step function. We interpret each rule as follows:

1. If a process has more than one neighbor then it waits for a message. Each message that is received results in removing the sender from the set of neighbors. Eventually, in an acyclic network, will the set of neighbors decrease to the point that the following step is activated.

2. If a process has precisely one neighbor c, we send to the neighbor the unit message and progress to the next state of $(\emptyset, \mathit{parent}\, c)$, i.e. we assign a parent and are waiting for a message back from the parent.

3. If a process has an assigned parent and receives a message for the first time, it can only be from a process that also has assigned this process as its parent: we thus indicate that we received a message and progress to the next step.

4. A process decides once it has received a message from its parent.

This rendition of the algorithm does not propagate back the decision of the tree edge back to its children. The author also implemented the variant of the algorithm which propagates back, but it is too complicated to give in this high-level outline.

With several of these algorithm definitions we can proceed with an overview of the simulation of the process description into a trace of its executions. We first need to consider various preliminaries:

Definition 3. *A* configuration $\mathscr{C}_G$ *of a network G is a pair labeling functions:*

1. *A function* $\lambda : V \to \sigma \vee \tau$ *that maps each process to either its current state or a decided result.*

2. *A function* $\kappa : E \to [\mu]$ *that maps each channel to a list of unreceived messages.*

A configuration transition *is a function that maps a configuration to another configuration.*

We are given an initial configuration: it is determined by the network topology and the initialization function given by a process description. We are looking for a particular configuration transition, namely one that is defined using a semantics that mimics the execution of distributed algorithms on real networks. We give the rules of our small-step semantics below, which is a single step in relabeling the network. Given the current state $\lambda(p)$ of any process p, either it is a state or it is a decided result. If it is a state $s \in \sigma$, evaluate the step function of the process description:

1. If the process result is **Receive**, and there is an unreceived message on a neighboring channel, i.e. $\exists q.\kappa(\langle p,q\rangle) \neq \mathit{nil}$, we remove the first message from that channel and apply the message to the receive function, of which the result becomes the next state of p.

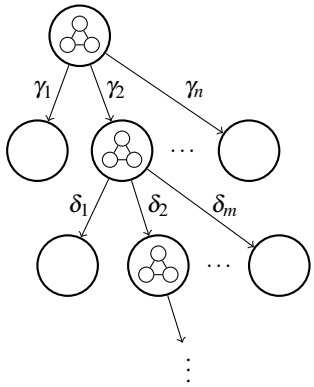
2. If the process result is **Internal**, we choose any of the resulting states. The next state of p becomes the chosen state.

3. If the process result is **Send**, we have some next state and a channel message: the list of unreceived messages of the corresponding channel is appended with the sent message, and the next state of p becomes the given state.

4. If the process result is **Result**, there is no next state of p and instead we label the process with the given decided result.

Note that in the current framework we only model FIFO channels. This can be lifted by adding another rule for permuting the channel messages by shifting the message on front to the back. Moreover, we have two sources of non-determinism in the way we have defined our semantics: choosing a process, choosing a neighbor if more than one neighboring channel has unreceived messages, and choosing any of the resulting state for the internal result. Finally, this evaluation serves as a definition of a configuration transition: given a configuration we apply these rules to obtain a new configuration.

In the implementation, we have defined not a single successor configuration, but a list of successors. The list monad is employed to produce all possible successors. This is best explained by visualizing the space of all configurations as follows:



We call this larger graph a simulation. For each node in the simulation, we have a certain configuration. Each transition, $\gamma_1, \ldots, \gamma_n$ is a possible successor from the initial configuration. Within such a successor we may have even more successors, $\delta_1, \delta_2, \ldots, \delta_m$, and so on. Each configuration transition generates a single event, a send event, an internal event, a receive event or a result event, which is then visualized, and for each configuration there may be multiple choices of selecting a successor.

Real-time updated simulation.

It is considered a nice feature to show the results of a simulation to the user as soon as possible. The tool is developed on an old machine, which immediately made clear that the simulation must be performed in a background worker thread to prevent the user interface from becoming unresponsive. This decision might be considered a nice feature, however, at a greater cost. The user interface code now has to deal with multi-threading coding to support commanding and interpreting the results from the simulation, possibly making the resulting code harder to understand than simpler single-threaded code. This cost can be directly observed as a decrease in tool quality, as the application may exhibit glitches or errors when the user controls the user-interface while the simulation simultaneously operates on the background, thereby changing the state of the objects as used in the interaction.

A solution for working around this decision is to restructure the core glue part of the application, between the user interface and the simulation framework. However, this was not performed during this project since that would delay the project.

User-controlled non-determinism.

The simulation framework implements a linearization of a computation, and at each point in generating the events for the simulation, may there exists multiple 'future worlds'. It was decided that the user must have control over this aspect of the simulation, thereby allowing users to explore not only a single computation but the full breath of the execution. However, this decision was made without carefully considering the possible interactions of the user with the system. This can be explained best by an example.

Suppose that the user wishes to reorder two concurrent events: the computation differs, but by definition is the execution the same. By moving an events, we have to account for natural constraints, such as a message send event must never happen after the corresponding message receive event. These constraints are fulfilled by giving feedback to the user that the move of the event is not possible, thereby honoring the constraint. However, with the introduction of user-controlled non-determinism, we directly expose

the linear simulation structure to the user: at each step in the simulation can the user determine what event is performed next, thereby changing the execution. The user might also have moved events, thereby breaking the correspondence between the representation of the timeline and the actual simulation, which is a linear sequence of events. Then it becomes not intuitive what choices are available at a given 'current time'.

Another problem occurs in the following scenario, in which two events that happen on the same process are swapped. The first event was sending a message to some other process, and in reply it sends back another message. The second event was the receive event that was immediately caused by the first event. Swapping these two events could make no sense: there is no execution where the receive events happens before the send event that eventually results in the receive event in the first place. It is an open problem for investigation to determine whether the user-facing constraints are sufficient in preventing these problems, or that other edge cases exists.

Although the interaction of swapping two arbitrary events was realized, it was not possible within this project to find an adequate solution to the latter problem. This problem is proposed as a future assignment in the last section.

Selection of algorithms.

Only a few algorithms are implemented. In particular, these algorithms all operate on undirected networks, where edges are FIFO channels. There are two kinds of algorithms implemented: centralized algorithms where the user has to supply a single initiator, and decentralized algorithms where the user has no say over the selected initiators. It was decided to limit the scope of the selection of algorithms within the development of this project, to prevent the project of becoming unrealized within the given time to work.

It is, however, quite conceivable to extend the framework to support the following notions:

- Directed networks, where not every edge is bidirectional. Currently, the directed network is implemented by default. A mode switch can be provided in the network component to allow users to create arcs, the simulation framework needs no adaptation.
- Weighted networks, where every edge has an assigned rational number. The network component can be extended to support a command for assigning weights to edges. The simulation framework needs only slight adaptation, namely in the representation of the assumption space, for algorithms that operate on weighted networks.
- Non-FIFO channels, where additional successor configurations permute any channel consisting of more than one message, by shifting the message from the front to the back of a channel.
- Unreliable channels, where additional successor configurations consists of a message that can be dropped for each unreceived message in a channel.
- Process crashes, where a process can unexpectedly crash. This needs several adaptations: the simulation must have at any configuration have a "crash" event as successor, thereby allowing the user to choose which process crashes at what time. Additionally, the configuration definition needs slight adaptation to extend either the labeling function from stateful or decided, to stateful, decided or crashed. Most algorithms that operate with crashing processes also expect failure detectors, which can be implemented by a special message that is broadcast to all neighbors that indicates that a process has crashed.
- Byzantine processes, where a process can send arbitrary messages and not follow its process description. This needs severe adaptations: the simulation must be extended, for each configuration, with an injection function for each byzantine process: the function accepts an arbitrary send event, that translates in that event being performed. The user must be able to construct any message from the message space, which requires a special component for synthesizing such messages by users.
- Dynamic networks, where the topology may change during the execution of an algorithm. Similar to process crashes, it might be useful to implement this by sending a special message to each process incident to a new or removed edge. Additionally, edges may be erased while messages are in transit, so this notion already requires the notion of unreliable channels. Indeed, if whole vertices may be removed and added, this notion is stronger than that of process crashes, since it furthermore allows processes to restart.
- Timed networks, where each process has additional state representing its local clock, and additional successor configurations allow the progression of this clock independently of the process description. It could even be the case that an extension could bound the execution time of a message receive, thereby limiting the possibility that a process indefinitely waits on a message that never arrives. The notion of timed networks is not explored by the author.
- Synchronous networks, with the additional assumption that there exists a global time, and progression of global time is only allowed after all processes are finished sending messages. This can be conceived as a special case of timed networks, where the progression rule applies globally.

It is the case that the following algorithms are implemented as part of the development:

- Tarry's algorithm, a graph traversing algorithm.
- A depth-first search algorithm, based on Tarry's algorithm.
- A depth-first search algorithm that tracks visited nodes, as a variation on the previous algorithm.
- Awerbuch's algorithm, a token-based traversal algorithm.
- Cidon's algorithm that is an optimization of Awerbuch's algorithm.
- A tree algorithm, already given above.
- A tree algorith with acknowledgement, as a variation on the previous algorithm.
- An echo algorithm, for broadcasting.

The implemented algorithms may not be optimal. For example, for Awerbuch's algorithm, the implementation uses the following state space:

- A Boolean variable whether a process has the token.
- A state, choice out of: received unseen + channel, received seen + channel, replied + channel, undefined, initiator unseen, or initiator seen.
- A set of channels to inform.

- A set of acknowledges channels,
- Maybe an intended channel,
- A set of forward channels,
- A set of info channels,
- Maybe a last channel,
- Maybe a pending acknowledgment channel.

The implementation for Awerbuch has furthermore 10 clauses for its step function. However, it seemed more important during development to have greater sense of understanding than finding the most optimal state space for expressing a certain algorithm.

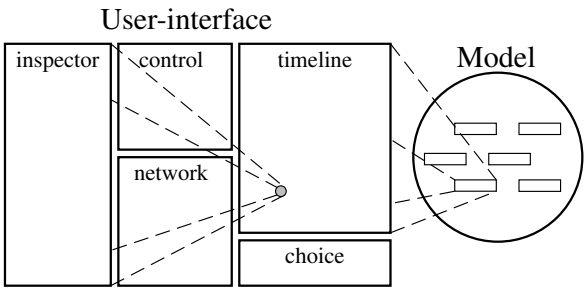
Inspecting detailed state.

While developing the algorithms, it was found very useful to have a state inspector. It allowed the author to step through the implementation of the algorithms and reason about the implementation. For any end user that uses the tool for the first time, the inspector component might be overwhelming: for instance, for Awerbuch's algorithm the previously listed state space is shown for every event and every process that can be inspected, without any context of what these values represent. However, the inspector might serve as a useful tool when implementing new algorithms, precisely because it compactly gives just a representation of the state space at some instant, which could account for the effort spent in developing this component.

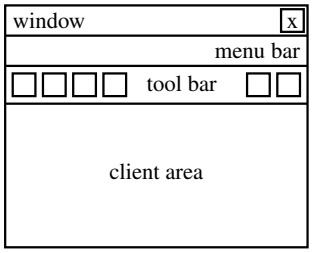
As a potentially downside of the current implementation, many objects referred to by the inspector component are static, i.e. not interactive. It might be useful in a future development to architect the inspector component as a more central part of the interaction. For example, if the current state of some process refers to a set of neighbors, that the user can highlight them in the network component.

3. USER-INTERFACE DESIGN

The user-interface is constructed in close relation with the aforementioned architecture. Each component is represented as a window, which peers into the underlying model of that component. The following picture shows the relation between windows and their underlying components: this is almost 1-to-1, except for the simulation component which has no immediate window. Instead, the results of a simulation are observable through all other windows: the timeline window shows all events and messages within the current execution, the choice window shows parameters for choosing executions, the network window shows messages in transit and depicts process states, and the inspector window shows detailed state information based on the selection of either events and messages in the timeline window or processes and channels in the network window.



There are three main windows: the control window, the network window and the timeline window. The other two windows are auxiliary: the inspector window and the choice window. The choice window directly depends on the timeline window, and its visibility depends on the user-input mode of the timeline window. The main windows have the following structure:



Here, each main window has a menu bar that contains a categorized list of actions that the user can perform. The tool bar contains only the most important actions the user may wish to access, divided in two categories: user-input modes and user actions. The user-input modes changes the interaction of the rest of the window, of which we will give two examples below. User actions are simply shortcuts for menu items, that make the actions more discoverable. Auxiliary windows have a simpler structure either lacking the menu bar or the tool bar or both. We now give a description of each client area.

Network client area.

The network window's client area consists of an interactive graph editor. The editor consists of two objects: vertices (corresponding to processes) and edges (resp. channels). The editor has four input modes: selection, add vertex, add edge and erase. Users can select any input mode. Additionally, the editor may be disabled to disallow any user modifications of the network topology during an active simulation, but still allows selection of the individual objects: it forces the selection mode.

The selection mode allows users to move vertices around, and select zero or more objects (either vertices or edges, either homogeneous or heterogeneous).

The add vertex mode allows users to add new vertices. Vertices are automatically labeled, by choosing for the new vertex the first free label from a predefined sequence of names: p,q,$\ldots z$, $pp, pq, \ldots, pz, qp, \ldots$.

The add edge mode allows users to add new edges and new vertices. An edge is created by dragging the mouse pointer from a vertex onto another vertex; if the mouse pointer is released without pointing to an existing vertex, a new vertex is created.

The erase mode allows users to remove vertices and edges, either by clicking on the objects or by dragging around and sweeping the area clean.

Control client area.

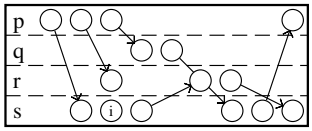
The control window's client area consists of a selection of implemented algorithms and parameters. Based on the assumptions of an algorithm, the user can either select initiators or not, by selecting the processes from the network window and assigning them as initiators. Finally, the client area shows based on the selected algorithm whether the network needs to be acyclic and whether the algorithm is centralized or decentralized.

The control window is the central control of the system. The actions for the control window are starting and resetting the simulation, or resetting the whole application to its starting state.

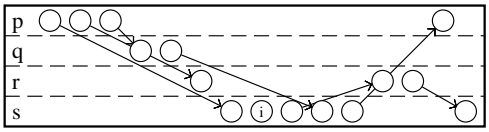
Timeline client area.

The timeline window's client area consists of a trace editor. The editor consists of a table where on the rows processes are shown, the columns represent an instant in time and time flows from left to right, and events are objects that happen at a particular time instant associated with precisely one process. Other objects are messages which connect multiple cells in the table. The editor has two input modes: concurrent and linear.

An example of a concurrent timeline is shown directly below. The four rows each represent a process. A concurrent timeline allows that within a single column multiple events can happen at different processes. For example in the second column a send event happens at *p* and a receive event happens at *s*.



The corresponding linear timeline is shown below. Note that a concurrent timeline may have multiple linear timelines: the user-interface only allows users to construct a concurrent timeline from a linear timeline. Whenever the timeline is concurrent and the mode is switched back to linear, the original timeline is reverted to the linear order which was generated by the simulation.



In the concurrent edit mode, users can drag groups of events. A group of events is defined as a sequence of events happening at the same process, starting with a receive event followed by as many as possible internal and send events. Grouping of events corresponds closely to the framework used for simulation: only when a process receives an event will the simulation consider events at possibly other processes.

The user is limited by a *happens before* constraint: for example, dragging a group of events past the sender of the first receiving event is disallowed, and similarly, dragging a group past a receive event corresponding to the send events of the group is disallowed. Thus, an invariant of the timeline is that all send events happen before corresponding receive events. The user receives feedback if this invariant is broken by the red coloration of the message arrows, and snaps each group back to its last permitted location if the user commits a dragging operation that is disallowed.

Additionally, the client area shows a time ruler, which indicates the currently visible point in time in the choice window and the network window. The user can either command the time to move one step forward or backward, or drag the time ruler around.

Part of the timeline client area is a swap gesture, that allows users to swap two groups of events by dragging them beyond each bounds. The gesture recognition and animation is implemented, but due to difficulties related to the open problem of the choices in the case of the concurrent edit mode, this feature has no effect on the simulation in the current implementation.

Choice client area.

The choice window's client area consists of a horizontal list of events, showing the selected simulated successor world. The user can modify the selection to influence which execution is simulated. These changes are immediately reflected in the timeline window. The choice window is only visible when the timeline is in the linear edit mode, to prevent showing choices not in sync with the current time as indicated by the time ruler.

It is still an open problem to represent choices even in the case of the concurrent edit mode of the timeline: see the last section for a more detailed description of this problem.

Inspector client area.

The inspector window's client area consists of a table with key value pairs. The keys may be nested to clearly indicate which category the corresponding value belongs to. The inspector changes the detailed information whenever a user changes the selected objects or the current time indicated by the time ruler in the timeline.

4. IMPLEMENTATION

The project consists of two main pillars: the Java code and the Haskell code. The table below summarizes the total number of source lines for each part:

	Java			Haskell
	GUI	Glue	Controller	Simulation
LOC	~7.5K	~2.5K	~3K	~800

The number of lines of code only include non-empty lines and includes comments. But comments are sparse: there are approximately 250 single-line comments, and no block comments. There are 4 commented out source regions, mostly dealing with testing code. Note that the Frege compiler translates the ~800 lines of Haskell code into ~13KLOC Java code.

The Java implementation is packaged in the following way:

daviz Contains the controllers for linking the GUI components and the Glue code together. This package results in an implementation of aforementioned architecture.

daviz.glue Contains a Java model for interpreting the Frege compiled results, but is algorithm agnostic. It contains translations methods for reading and preparing data structures that are consumed by the Haskell implementation.

daviz.glue.alg Contains algorithm-specific glue code. Each object in this package encapsulates an implemented Haskell algorithm, and provides methods for loading and unloading into the expected format and extracting detailed state information for display within the inspector window.

daviz.sim Contains the compiled Haskell code, i.e. Java classes that are the output of the Frege compiler.

daviz.images Contains icons (16x16 pixels and 32x32 pixels) for display in tool bars and menu bars in the GUI.

daviz.ui Contains model interfaces and default model implementations and GUI components. For example: JCarousel for the choice window, JAssignmentField for selecting initiators in the control window, JCoolBar for showing tool bars, JGraph for the network editor, JTimeline for the timeline editor, JKnob used by both JGraph and JTimeline for draggable events or vertices, JStatus for the status bar, and JInfoTable for the inspector window. These components are written with reuse in mind.

daviz.ui.plaf Defines the UI interfaces for complex components, such as JCarousel, JGraph, JInfoTable and JTimeline.

daviz.ui.plaf.basic Implements the UI interfaces for rendering, handling user input, recognizing input gestures and updating component models based on user-input.

The GUI components are implemented according to standard Swing practices. Each component is separated in three parts: a model (an interface and default implementation), a controller (classes prefixed with J), and the view (a UI interface and the basic implementation).

The Haskell implementation is straight-forward:

Set Provides a list-backed set data type.

Graph Provides a graph implementation, where edges are pairs of vertices and a graph is a set of edges.

Process Provides a framework for defining process descriptions.

Event Provides a framework for working with event traces, i.e. the paths in a simulation.

Simulation Provides a configurations and a framework for computing the simulation tree.

Awerbuch, Cidon, DFS, Echo, Tarry, Tree, TreeAck, Visited Provides the implementation of algorithms.

The development does not include any automated tests for checking properties of the implementation. This has two reasons: first, GUI code is hard to test. Slightly changing the way a component works immediately breaks existing test cases, and hence is slower to develop the GUI. Second, the simulation back-end was tested by performing a benchmark (called RandomProgram) that simulates a certain algorithm implementation and chooses random executions along the way. This program helped to ensure the simulation is robust: the Java glue code was tested by 'fussing' random networks and translating it into the Frege data structures, while the Haskell implementation was tested against underspecification (i.e. partial functions). This benchmark ran for 5 days without any reported errors.

The implementation has acceptable performance for use in a visualization tool, because it is developed on hardware older than 8 years with acceptable performance to ensure that it also runs with similar or better performance on newer hardware. The software is tested on Windows 7, Windows 10 and GNU/Linux (Fedora) operating systems. The software depends on Java 1.8 Standard Edition run-time to be installed on the target machine.

5. CONCLUSION

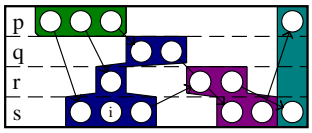
The author acknowledges prof. Wan Fokkink for his teachings of the Distributed Algorithms course on VU University Amsterdam, and his supervision and patience with this project. Special thanks to Jacco van Splunter for giving practical sessions of Distributed Algorithms, and his feedback and suggestions during the project. This document is typeset using LaTeXand the ACM SIG Proceedings template.

5.1 Future Assignments

This section contains short descriptions of tasks that may improve the software. These tasks are originally conceived as part of the project, but were not realized due to time constraints. Of course, anybody who appreciates these tasks may accomplish them in the future, or anything else he or she feels that would improve the software.

Visualizing Concurrency Classes.

A concurrency class contains all events that are concurrent. Two events are concurrent whenever they are independent, that is, if there is no restriction by the *happens before* relation that one must occur later than the other. An example that shows concurrency classes on a timeline is given below. A possible direction for implementing this feature is to: 1. separate the linear and concurrent order of a timeline, i.e. a concurrent order is a "view" of a corresponding linear order that is output by the simulation, 2. normalize the linear order by the normalization procedure for Trace monoids, 3. add visual elements to the timeline component that outline each group, and make sure that the same group between processes are visually connected. Alternative direction: implement Leslie Lamport's logical clock. Each clock value defines its own concurrency class.



An important assumption for working with concurrency classes is the notion that all locally grouped events (as mentioned earlier: that starts with a receive event and contains as many internal and send events possible) all happen instantaneously. Hence, the timeline editor does not correspond 1-to-1 to this notion, since events within a group also take up column space, thereby not allowing all events within the same concurrency class to commute freely.

More algorithms.

Obviously, the software can be extended with more algorithm implementations. Many algorithms have different assumption classes, for example some expect data values to be initially present, others expect weighted edges, et cetera. Thus, adding more algorithms also requires adapting the visualization tool, perhaps adding new attributes to the network editor for containing such assumptions. This is a hard assignment, because one has to carefully select comparable algorithms before implementing the changes to the framework and the visualization tool. Furthermore, this assignment cannot be worked on by two independent authors if they choose to work on different problem sets, since the two adaptations may be hard to integrate back into the same software. However such a parallel endeavor might result in a useful discussion afterwards, for analyzing the different requirements of the framework and the tool, and comparing how they can be combined in one generalized model.

Swapping gestures / Concurrent choices.

Originally, the author conceived the notion of swapping two event groups that happen within the same process. In some cases it may be possible to actually perform such a swap, in other cases it is not. This highly depends on the algorithm that is simulated: swapping two event groups may drastically change the future execution, and even the existence of one of the participating groups may disappear, or the events comprising the least recent event group may alter.

It is expected to be a challenge of finding a way to handle all these subtle cases in the timeline component. It also requires a notion of identifying events between different executions, namely to keep track of the dragged event groups and the events within. Identification may depend on the particular simulated algorithm, and thus may require an abstraction that is realized for each algorithm independently.

Related to this problem is to find a meaningful way of showing choices when events are reordered in the concurrent edit mode. One needs to find a way to map the concurrent events back to their original place in the simulation. When selecting a choice in the choice window, the simulation regenerates all events after the chosen event.