# DaViz – Distributed Algorithms Visualization Tool

## Individual Systems Practical – 2019-2020 Edition – Technical Report

Wesley Fu-En Geniz Shann
Developer
Vrije Universiteit Amsterdam
Master Computer Science
Track Internet & Web Technology
w.genizshann@student.vu.nl
2639155

Wan Fokkink
Supervisor
Vrije Universiteit Amsterdam
Computer Science Department
Theoretical Computer Science
w.j.fokkink@vu.nl

***Abstract*—*Context*: DaViz is an e-learning prototype tool for the visualization and simulation of distributed algorithms. Implemented in Java and Frege, the tool was developed to support the lectures for the Distributed Algorithms course taught at the Vrije Universiteit Amsterdam. The prototype, although functional, required some improvements.**
***Goal*: Therefore, this project has four goals to improve DaViz: (1) propose a lightweight port of DaViz without external dependencies (i.e. Frege), (2) improve the development process, (3) support and simplify the collaboration between multiple developers (4) fix minor issues, ensuring that end-users (students) can use all features available in DaViz.**
***Method*: The first goal is facilitated by abstracting the existing simulation module of DaViz, extracting the common implementation to a shared module between the native Java port and the Frege port. Afterwards, only the specific implementation needs to be implemented in the Java port (i.e. algorithm specification and event generation). The second goal is achieved by stabilizing the development environment and aiming toward a zero-configuration setup. The third goal is achieved by adopting the usage of version control system GitHub and its features, such as issue and pull request tracker, releases and tag versions and development branches. The fourth goal is achieved by identifying the root of existent issues and implementing suitable solutions.**
***Conclusion*: Majority of the contributions of this work are aimed for developers – stable development environment, zero-configuration, versioning control, reduced complexity of the simulation module. For the end-user, there is a lightweight prototype port of DaViz fully implemented in native Java, the addition of the Echo algorithm for simulation in the Frege port and correction of the Tree algorithm.**

***Index Terms*—Distributed Algorithm, Simulation, Vizualisation, Design Patterns.**

## I. Introduction

DaViz is a tool to improve the learning of distributed algorithms by allowing students to simulate the computation of diverse algorithms in a given network topology. DaViz prototype was developed in 2017, motivated by the Distributed Algorithms[1] course at the Vrije Universiteit Amsterdam[2], to be used as a supplementary tool that students can use to further understand the lecture concepts and algorithms.

However, the initial prototype developed is still in the early phases, lacking in terms of features and contents for the end user. For instance, in the current version, the tool contains only distributed algorithms categorized as wave algorithms. Additionally, there is no support for developers to maintain and contribute to the development of the tool.

This report's goal is to describe a series of improvements and corrections performed in the tool, for both end-users (students) and developers. By describing all changes performed in the software and possible future works, the target audience of this report are developers interested in improving DaViz.

The remainder of this report is organized as follows. The features, architecture and state of DaViz are specified in section II. Next, the project-specific goals are described in section III, followed by the improvements performed in DaViz in section IV and the software package in section V. In section VI, similar tools are briefly described. Lastly, the project achievements are summarized in section VII, where possible future work is described.

## II. DaViz

DaViz consists of two main pillars: simulation and visualization. The *visualization* was implemented using the Java[3] programming language and Swing, the "standard Java implementation for graphical user interface". The *simulation* was implemented using the Haskell[4] programming language and Frege[5], a "Haskell for the JVM", which allows generating Java classes from the Haskell implementation. Given the nature of Java and Frege, in principle, DaViz can be used and developed in all major desktop platforms: OS X, Windows and Linux.

### A. Usage and UI

A usage example of DaViz simulating the Tarry algorithm is illustrated in Figure 1. The UI consist of 3 main windows – Control (DaViz), Network and Timeline – and 2 auxiliary windows – Information and Choice. In the Control window, the user can select the algorithm to be simulated, verify the requirements in the network topology that must be satisfied – cyclic/acyclic and centralized/decentralized – indicate the initiator process and start the simulation. In the Network

[1]https://studiegids.vu.nl/en/2019-2020/courses/X_400211
[2]https://www.vu.nl/en
[3]https://www.oracle.com/java/
[4]https://www.haskell.org/
[5]https://github.com/Frege/frege

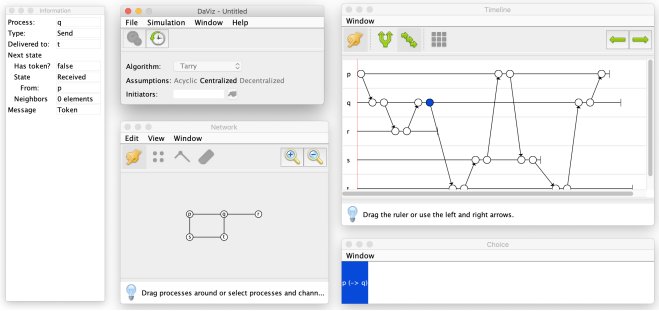

Fig. 1: DaViz simulating Tarry algorithm.

window, the user can define the network topology and select one or more initiators. Once the network topology is defined and the simulation is started, one possible computation is presented in the Timeline window. Here, send and receive events are described for all processes in the network. The user can click in each event and its details are presented in the Information window, such as the event type, the sender, the receiver and the message. As different algorithm have different designs, the fields in the "next state" can be different. The user can move the stage of the computation forward and backwards in the Timeline window, by manually moving the red line or clicking in the left/right arrows. This will update the data in the Information and available options in Choice windows. The Choice window presents all available choices the algorithm can do in the current stage. There will always be at least one choice available, otherwise, the simulation would fail to terminate. If the user opts for another choice, the Timeline is updated accordingly to reflect the new computation.

### B. Structure

Both DaViz pillars are implemented as an Eclipse[6] project. The simulation project contains only a sim package with the Frege implementation, which is compiled to Java classes using the Frege Eclipse plugin[7]. The visualization project contains four packages: UI, images, daviz and glue. The first implements all the Swing components required for the visualization and the second provides the assets used in the UI. The third package contains the classes responsible to manage the application and to structure the windows in the UI. The fourth package is responsible to provide a bridge between the visualization (daviz package) and simulation (sim package) projects. The Frege library, imported as a Java Archive (JAR) file, is used by the glue and sim packages.

Figure 2 illustrates this structure, where the pillars are represented in blue panels, the Java packages in green panels and the Frege library JAR in orange. The packages hierarchy is also illustrated, where daviz is the root package containing all other packages. Note that both UI and glue packages have sub-packages omit in this diagram for simplicity reasons. Similarly,

[6]https://www.eclipse.org/

[7]https://github.com/Frege/eclipse-plugin

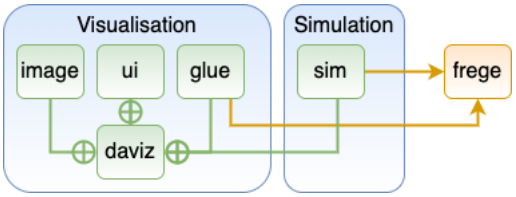


Fig. 2: DaViz original structure and Frege dependency.

only the package dependency towards the third-party library is displayed. Internal package dependency is omitted.

### C. Current status

Currently, the tool lacks some features in terms of user experience. For instance, there are no mechanisms to save the state of the network topology of a given computation. It is not possible to undo/redo changes in the topology either. An interesting feature that is not available is the possibility to see in "real time" (*i.e.,* the current stage selected in the timeline window) changes during the simulation reflected in the network topology defined in the network window. For example, in the Tarry algorithm simulation illustrated in Figure 1, relevant information to reflect would be already visited process, the process or the edge with the token and the progress of the spanning tree.

In addition to UX features, the tool does not contain a full set of algorithms, instead, there are a few selected wave algorithms only: five traversal algorithms (Awerbuch, Cidon, Tarry, DFS and DFS with visited checking), two Tree algorithms (with and without acks) and one Echo algorithm. From these 8 algorithms, the echo algorithm is currently disabled and both tree algorithms cannot be simulated due to incorrect identification of cycles in the network topology.

Another issue in terms of development is the usage of Haskell. Developers would be expected to be familiar with developing both Java and Haskell applications to be able to fully understand the DaViz implementation. While Java is a popular object-oriented programming language, Haskell might not be as widely used. Given the nature of the Haskell functional paradigm, it is expected that developers not familiar with it will have some difficulties in getting started in the development of the simulation. From a more technical point of view, the usage of Haskell makes it more difficult to implement control algorithms (*e.g.,* deadlock and termination detection, etc.) in the top of computational algorithms (*e.g.,* wave, routing, etc.). That is because Haskell (and in particular Frege's implementation) do not offer well-defined support to Java's runtime type reflection API, a feature that allows a Java application to examine and modify internal properties of a class, method or interface during runtime [1].

Currently, the DaViz source code implementation is not available in any versioning control, such as git. Instead, it is available on the author homepage [2], together with the system design goals and technical report. This makes it difficult for interested developers to contribute to the development of DaViz and synchronize between multiple developers.

## III. Project Goals

### A. Improving the development experience

This goal refers to a series of changes in the development process to improve the development experience and consequently improving the quality of DaViz.

*1) Versioning control and collaboration:* The first step towards improving the development experience is to create a repository in GitHub to host the source code and to provide version control. This will allow developers to contribute to the tool development and to collaborate with other developers. The usage of the issues and pull-request features will allow understanding each change made in the software. To ensure that a pattern is followed, templates for pull requests and issues will be suggested. To facilitate the distribution of DaViz for the end-user (students), the releases and tagging feature of GitHub will be used. Improvements in the README and getting started instructions will also be considered.

*2) Development environment and configuration:* The usage of the Eclipse Frege plugin can cause instability in the development environment. There are cases in which the Frege source fails to compile and throws several errors. Furthermore, due to the nature of the plugin and how Eclipse itself manages its dependencies, it is not possible to share configurations between collaborators. The reason is that the Frege plugin downloads its files to the local development machine and the Eclipse dependency configuration stores the target absolute path. Currently, there is no workaround to the instability of the Frege plugin and collaborators must override the configuration themselves to work in DaViz. Therefore, this sub-goal is to make the development environment as stable as possible and allow the dependency configuration to be shared among collaborators.

*3) Explain the implementation of Frege algorithms:* This sub-goal targets collaborators that are interested in improving the Frege implementation of DaViz. By describing the implementation of a few algorithms in Frege, there will likely be a reduction in the time for collaborators to understand the implementation, especially for non-Haskell developers.

### B. Fix unavailable wave algorithms

As described in the section II-C, both Tree algorithms incorrectly identify the presence of cycles in the network and the Echo algorithm is disabled and thus not available for simulation. Thus, this goal is to ensure the correctness of the existing wave algorithms and allow users to simulate them.

### C. Implement a Java port for the simulation module

The third and main goal of the project is to implement a native Java port of the simulation module. There are two reasons for supporting this port. First, it simplifies the implementation of control algorithms on top of basic/computation algorithms. Second, it reduces the complexity of DaViz, allowing to distribute DaViz without external dependencies and allowing non-Haskell developers to contribute to the project without requiring them to learn Haskell.

| Version | Description |
|---|---|
| $v0.1.0$ | The original DaViz prototype. |
| $v0.1.1$ | Fixes for the Tree and Echo algorithms. |
| $v0.2.0 - rc.1$ | Major refactoring to support a native Java implementation of the simulation module. |
| $v0.2.0 - rc.2$ | First prototype of DaViz using the native Java port. Contains only Tarry traversal algorithm. |
| $v0.2.0 - rc.3$ | Fixes a bug in Tarry where computation might fail. |

TABLE I: Released versions of DaViz.

Note that this native Java port will not replace the Frege port as the DaViz simulation project, instead, developers and the end-user will have the option to select one of the ports. Both ports must be independent and provide the same API for the visualization project to consume. The idea is to design DaViz components in a plug-and-play approach, where developers can easily switch between ports. Similarly to goal III-A2, the result of adding a new port should not introduce additional configuration to set up DaViz in a local development machine.

## IV. DaViz Enhancements

The steps and changes to achieve the goals defined in the previous section are described next.

### A. Improving the development experience

*1) Versioning control and collaboration:* To support effective collaboration between developers, a version control system is required. For this purpose, we opt to use git[8] and GitHub[9]. Git is an open-source, simple and elegant distributed version control system and GitHub is a free, popular and well-known solution that provides hosting for software development using git.

We adopt the popular git branch model first introduced by Driessen [3]. Currently, only the master, development and features branches are in use, however, it would be desired to fully use the model and ensure that it is followed. For the context of tagging and releases, Table I describes all five versions available at the moment of writing this report. Goals III-A3 and III-B are accomplished in version $0.1.1$ and goal III-C in version $0.2.0 - rc.1$ to $0.2.0 - rc.3$.

To make clear what are the added features and fixed bugs in DaViz, the issue tracker and pull request features are used. To convey with more clarity the context and goal of new issues and pull requests, templates are suggested based on repositories of popular frameworks, such as Angular[10] and React[11]. There are currently three suggested templates for new issues – bugs, features and refactoring – and one suggested template for new pull requests. Since DaViz cannot be considered a big application and is not yet widely used, such well-defined patterns and usage are not a major requirement at the moment,

[8] https://git-scm.com/
[9] https://github.com/
[10] https://github.com/angular/angular
[11] https://github.com/facebook/react

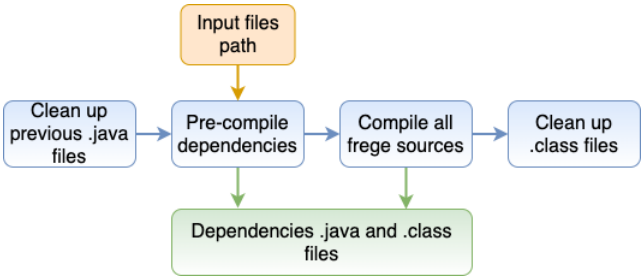


Fig. 3: Frege builder script pipeline.

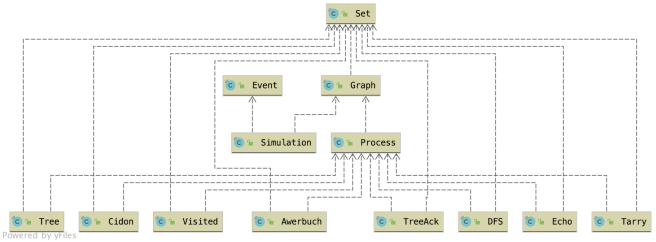


Fig. 4: Dependency graph of Frege implementation of DaViz.

however, enforcing these aspects will certainly help future collaborators.

*2) Development environment and configuration:* A simple solution to avoid issues with the absolute path could be storing the JAR in the project directory and targetting a relative path. However, the Frege plugin is configured to target the initial location outside the project directory. Instead, we decided to replace the plugin by a simple semi-automatic script that compiles the Frege sources. Figure 3 illustrates the basic pipeline of the Frege builder script. Blue panels refer to the pipeline steps, the orange panel refers to manual input required and the green panel refers to the output generated.

The first step is to remove all Java classes generated from the Haskell sources in the previous build (see Figure 4 to verify which classes are removed). Note that this step occurs only in the com.aexiz.daviz.frege package, which should contain only Frege implementation and no manually added Java class. The second step is to compile all Frege sources that are used in another Frege source. Note that Frege builds both Java classes and executable bytecode. The third step compiles all Frege sources in the codebase, including the sources compiled in the previous step. The fourth and last step is to remove all .class files generated. This last step is required because the Frege sources are compiled into the src directory. This is done to address unknown references to the compiled type. Although the application could still be executed if outputting the built files to the default output directory, the IDE would not recognize the compiled classes in the development stage, resulting in unnecessary noise in the IDE error messages window. Thus, this last step is performed to clean up the src directory.

In big codebases with several dependencies between classes, the manual input required in the second step would certainly add a non-negligible overhead in the development process. However, the Frege implementation of DaViz is rather small. Figure 4 illustrates the dependency graph between the implemented Frege sources in the current version of DaViz. There are five Frege sources that specify the core definitions for simulating algorithms and eight Frege sources that specify each algorithm implementation. From this total of thirteen sources, there are only four sources used by another Frege source and must be informed in the second step of the pipeline: Set, Graph, Event and Process.

Given that the plugin requirement was dropped, Eclipse is no longer mandatory. After searching alternatives that provide more flexible dependency configurations and a stable development environment, we opt to replace Eclipse with IntelliJ IDEA[12]. In addition to these benefits, the change also facilitates achieving goal III-C due to how IntelliJ IDEA organize projects. Instead of having multiple projects that depend on each other, the IntelliJ approach is to create a single project that has multiple modules. Therefore, instead of having one project for the simulation and one project for the visualization, we create a DaViz project and make one module for each pillar. This simplifies the dependency management and allows to modularize the project as needed, which is needed to achieve goal III-C.

Additionally, it offers support to easily share common configurations between collaborators. The configurations shared in the repository are the following:

- Dependency between modules;
- Running frege build script;
- Running DaViz application;
- Build artefact (JAR) for DaViz;

For convenience, additional configurations are added to run DaViz using the Frege or Java port and running the Frege build script before running DaViz. Similarly, since IntelliJ requires indicating the interpreter to be used to run the scripts, additional configurations are added to run the scripts using Zsh and PowerShell. Although the build script was not tested in Windows OS, the configuration will likely work.

If desired, contributors can define their configuration to run DaViz using another interpreter or adding other scripts in the execution pipeline without sharing the files in the remote repository or breaking the feature for other developers.

*3) Explain the implementation of Frege algorithms:* The description of the Frege implementation of the Tree and Echo algorithm is added directly in the source code as inline comments. This description can be verified in PR #8[13].

### B. Fix unavailable wave algorithms

*1) Echo:* To verify the correctness of the implemented Echo algorithm, it was enabled in the visualization module and simulated using the Echo example described in the reference book [4, p. 24–25]. The computation described in the book was compared with the result displayed in the Timeline windows

[12]https://www.jetbrains.com/idea/
[13]https://github.com/praalhans/DaViz/pull/8

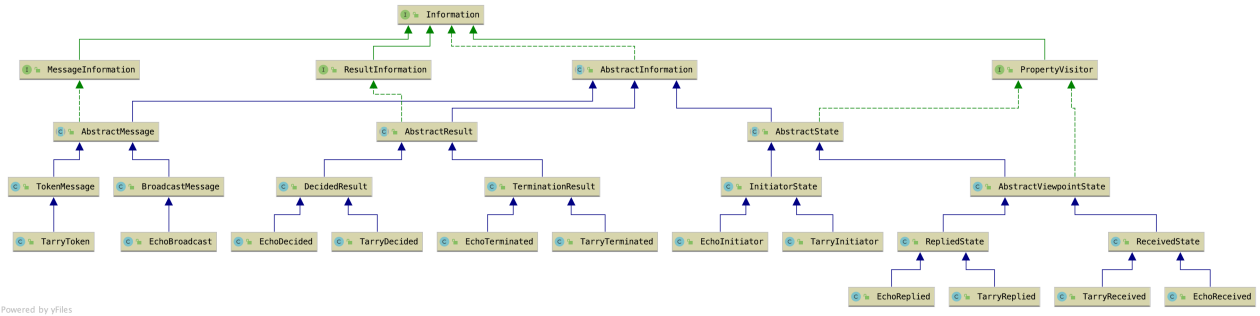


Fig. 5: Information type hierarchy.

and the information provided in the Information windows. To ensure correctness in different settings, the Echo algorithm was simulated varying the network topology and selected choice. As all executions were valid, therefore it is concluded that the Echo implementation present in DaViz is correct. The original author of DaViz was asked if there was any issue encountered in the implementation, opting to disable it. After that, the Echo algorithm status was set as working and ready to use.

*2) Tree:* The first step is to identify the root cause of the Tree algorithm incorrectly identifying the presence of cycles in acyclic networks. To accomplish this, DaViz was executed in debug mode and the execution flow is analyzed. This process revealed that the assumptions checkers in the visualization module were not fully implemented. While the centralized/decentralized checker was implemented, the acyclic checker would always return true. Since in the current version of DaViz the Tree algorithm is the only acyclic algorithm, it was the only one to throw the error.

After identifying the root cause of the error, implementing an acyclic checker in the visualization module is trivial. Given the target audience of DaViz and the expected scope of the simulation, we do not expect huge and complex networks to be simulated. Which means that the acyclic checker requires no optimization and has no constraints in terms of space and time. Therefore, a simple DFS approach is sufficient for implementing the acyclic checker.

Upon finishing the implementation, the correctness of the Haskell implementation of the Tree algorithm needs to be verified. Similarly to the Echo algorithm, the Tree algorithm is validated by comparing the result of the simulation with the examples described in the reference book [4, p. 23–24]. The evaluation also consisted of additional executions varying the network topology and execution choices. The comparison shows that the Tree implementation of DaViz is correct. For more details, see PR #4[14].

### C. Implement a Java port for the simulation module

Effectively achieve this goal can be split into three steps. First, all classes in the visualization module that have some dependency on the Frege library are moved to the simulation module. More specifically, the glue package illustrated in figure 2 and all its sub-packages are moved to the simulation module. Next, the simulation module is abstracted by extracting to a core simulation module all part of the Java implementation that have no direct dependency on the Frege library. The last step is to implement the actual native Java port of the simulation and to provide one algorithm for simulation. While the first step is straightforward, moving files to a new location, the second and third steps are described next.

*1) Abstracting original simulation module:* To abstract the simulation module, we use the ideas of the Abstract Factory creational design pattern [5], [6]. This pattern, common in the Java programming language, allows specifying common characteristic between objects without specifying its concrete implementation. In other words, an abstract interface is created to specify the common methods that a concrete class should implement. In the context of DaViz, abstract interfaces and common concrete classes that can be shared are created in the core module and specific concrete classes are created in the native Java and Frege ports, each specifying how to simulate algorithms.

An interesting target for the Abstract Factory design are the classes related to Information. In the DaViz context, information can refer to message information (*e.g.,* token, info, ack, etc.), result information (*e.g.,* termination, decided), state inform (initiator, received, replied, etc) or property information (describes the data in the Information window). Each specific algorithm must implement a concrete class to define all this information. In the original design of DaViz, each algorithm class would redefine these properties, repeating the same implementation multiple times. Thus Information and its child implementation make a perfect target for the Abstract Factory design.

Figure 5 (partially) illustrates the result of applying the extended Abstract Factory design pattern in the Information classes. Interfaces are represented in green and classes in blue. Dashed lines refer to classes implementing a given interface. This hierarchy has five layers: the root interface, the specific interfaces types, the abstract classes, the concrete classes and the specific algorithm concrete class.

Information interface is the root specification, which is extended for each of a more specific type of information (*e.g.,* MessageInformation, ResultInformation, etc.). Each

[14]https://github.com/praalhans/DaViz/pull/4

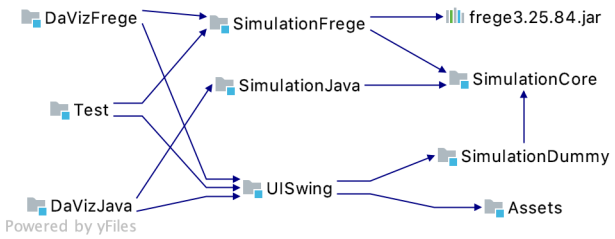


Fig. 6: DaViz structure and Frege dependency after refactoring.

interface is implemented by an abstract class that defines the common behaviour of that type of information (e.g AbstractMessage, AbstractResult, etc.). Note that there are two types of state information in DaViz, AbstractState specifies simple states within a process (*e.g.,* InitiatorState , UndefinedState, etc.) and AbstractViewpointState specifies states that are attached in a channel (*e.g.,* RepliedState, ParentState, etc.). Each abstract class is extended by a concrete class that specifies its exact behaviour (*e.g.,* TokenMessage, BroadcastMessage, DecidedResult, etc.). To finalize, each algorithm extends the concrete classes it needs (*e.g.,* TarryToken, EchoBroadcast, TarryDecided, etc.)

Although the last step could have been skipped and each algorithm would create a RepliedState instead of specifying their state (*e.g.,* TarryReplied, EchoReplied, etc.), it was decided to be specific on the design of Information classes for two reasons. First, the original design of DaViz already specified information classes per algorithm. Second, when implementing a control algorithm on top of a basic algorithm, specifying this level of details would surely be beneficial to distinguish between algorithms within a simulation.

The hierarchy from the Information type is defined in the Core module and are instantiated by the Frege and native Java simulation modules without any additional implementation.

Figure 6 shows the resulting module organization of DaViz after completing the refactoring/abstraction process. While the actual implementation of the original visualization module is not modified, its contents are split into three modules: UI Swing, Assets and Test. The Assets contains the image package, the Test contains the two tests classes RandomProgram and TestCases and the UI Swing contains all remaining classes from the original visualization module (except the glue package, which was moved to the simulation module).

The contents of the original simulation module with the addition of the glue package are abstracted and defined in three modules: Simulation Core, Simulation Frege and Simulation Java. As described in the previous sections, the core module specifies common behaviour and is used by both Java and Frege modules, each specifying specific behaviour on how to simulate algorithms in native Java or in Frege.

Note that the UI Swing module has no dependency to either simulation modules, instead, we introduce a Simulation Dummy module that simply exports classes directly used by the UI Swing module. At the moment, the Dummy module specifies only the DefaultNetwork, DefaultSimulation and Algorithms classes. The network and simulation classes are used to configure the simulation and start it, while the algorithms class exports to the UI a list of available algorithms for simulation. Extracting from the visualization module to the simulation module all classes that depend on Frege allowed to encapsulate the dependency on the Frege JAR only in the Simulation Frege module.

Lastly, the starting point of DaViz are now defined in the DaVizJava and DaVizFrege modules, each invokes all modules necessary to start the given module. Since the Frege library is used only by the Simulation Frege module, when loading DaVizJava, no external dependency is required. To generate the artefacts JAR, the dependency configuration used is the same for the dependency loaded for the DaVizJava and DaVizFrege modules. Details can be verified in PR #16 [15].

*2) Native Java implementation of the simulation module:* It was possible to understand the execution flow and internal structure of the simulation module in the process of abstracting its common features. Moreover, the majority of the implementation could be extracted to the Core module and reused in the Java port. This allowed implementing the new port with reduced effort and size.

Figure 7 illustrates the main execution flow when starting the simulation using the Frege port. Note that several transitions (mostly in the UI) were omitted to simplify the sequence diagram and focus on the most import flow.

When the user clicks in the “Start simulation” button in the Control Window, the action listener defined in the ControlFrame class is executed. It simply invokes the Controller class, that does three tasks. First, it verifies if the network topology defined by the user complies to the selected algorithm assumptions. If one of the assumptions is not satisfied, the execution is aborted and an error window is displayed to the user with the violated assumption. If all assumptions pass, it parses the network topology into a format known to the Simulation module. The last task it does, which is omitted from the diagram, is requesting the DefaultSimulation class to load the initial state of each process. Next, the SimulationManager is invoked and make the last preparations to start the simulation and receive the results. In this step, a loop is started, requesting the next execution to the ExecutionStepper class.

Here, the execution moves from the UI module to the Simulation module. Starting in the DefaultExecution class, by receiving a request for the next execution. As it is Frege nature to only compute when the data is requested, the first step is to run the Frege implementation for the next possible step of the execution (*e.g.,*send a message, terminate, etc.). This occurs in the unloadSucessors method, in the form of a loop while there are valid possible steps (*i.e.,*possible options in the Choice window). If a successor is found, a DefaultExecution object is created as a wrapper of the

[15]https://github.com/praalhans/DaViz/pull/16

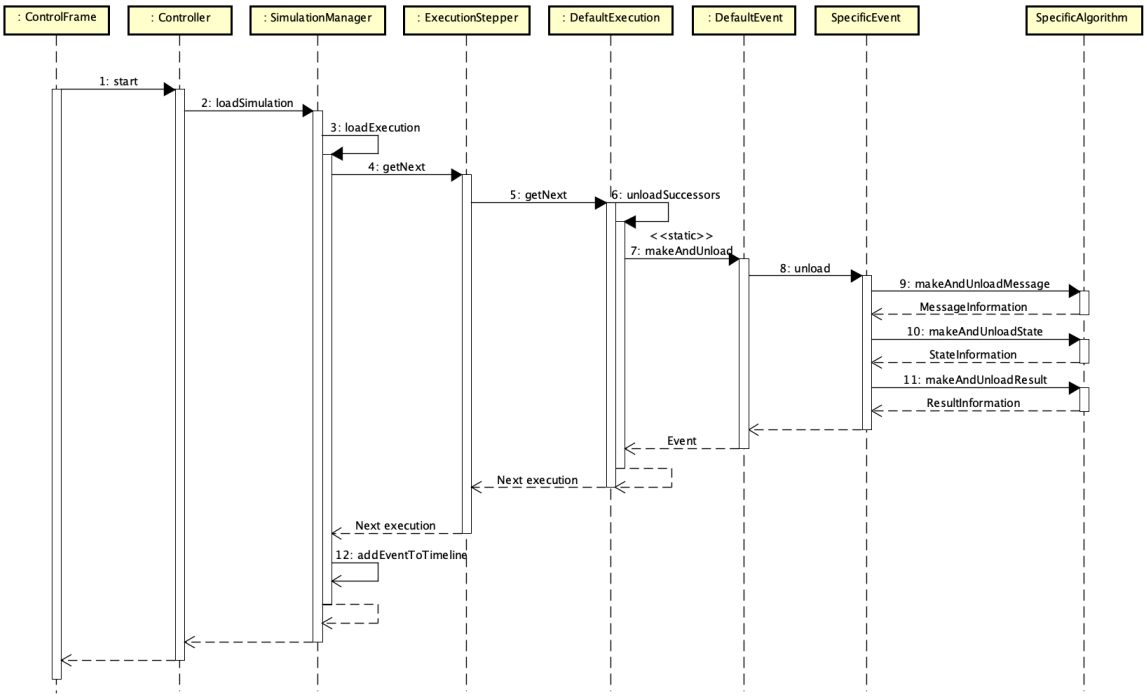


Fig. 7: Main execution of the simulation in the Frege port.

possible step and the DefaultEvent class is invoked to create a wrapper object for the event that occurs in this step.

The DefaultEvent class determines the type of the event (*e.g.,*send, receive, etc.) and request the appropriate event class to extract the information. For instance, a send or receive events would extract two types of information. First, it would invoke the method makeAndUnloadMessage to extract what is the message in the channel (*e.g.,*token, broadcast, ack, etc.). Then the method makeAndUnloadState to extract the next process space state that a given process will have upon sending or receiving the message. A result event would invoke the makeAndUnloadResult to extract a termination or decided information, depending on whether the process is the initiator or not. Lastly, an internal event would only extract the process space state information.

Next, the first wrapper object of the execution is returned to the UI and this process is repeated until the simulation is completed.

In the Frege port, "make" and "unload" methods refers to executing the Frege implementation and parsing the results to a known structure in the Java classes. The empty returns in the simulation module mean that information computed was set directly in the class attributes.

The implemented native Java execution flow, illustrated in Figure 8, is slighter simpler, essentially because there is no need to invoke the Frege execution and parse the results to a known Java structure. The four classes from the UI module show the same flow as there were no significant changes in the UI.

The differences start in the DefaultExecution class, where an entire set of possible executions is requested to the specific algorithm class. Next, the algorithm class iterates over the processes space states, verifying for each process if a new event can be created. For instance, if the last step was a process $P$ sending a message to a process $Q$, then a receive event can be created in the process $Q$. All created events are stored to a list of possible events. There should be at least one event, otherwise, the simulation fails to terminate. Upon finishing the iteration on the process space state from all process, the event list is returned to the DefaultExecution, where a DefaultExecution object wrapper is created for each event in the list. Similarly to the Frege port, the first execution object is returned to the UI.

In the Tarry implementation, in addition to the process space state, it maintains a flag to inform if the token is in transit or with a process and which process will receive the token next. The information regarding the token was introduced to avoid processes moving to a received state before the token was even sent to the channel.

While in the Frege port, the DefaultEvent class is used as a bridge between the DefaultExecution and the specific events and algorithms classes, in the Java port, the DefaultExecution communicates directly with the specific algorithm class, that in turn generates a specific event. The fact that the event classes are specific to ports prevents modelling them as common classes in the Core module.

Details can be verified in PR #28[16].

In the current version, the native Java port of DaViz allows simulating only with the Tarry algorithm. Similarly to how the

[16]https://github.com/praalhans/DaViz/pull/28

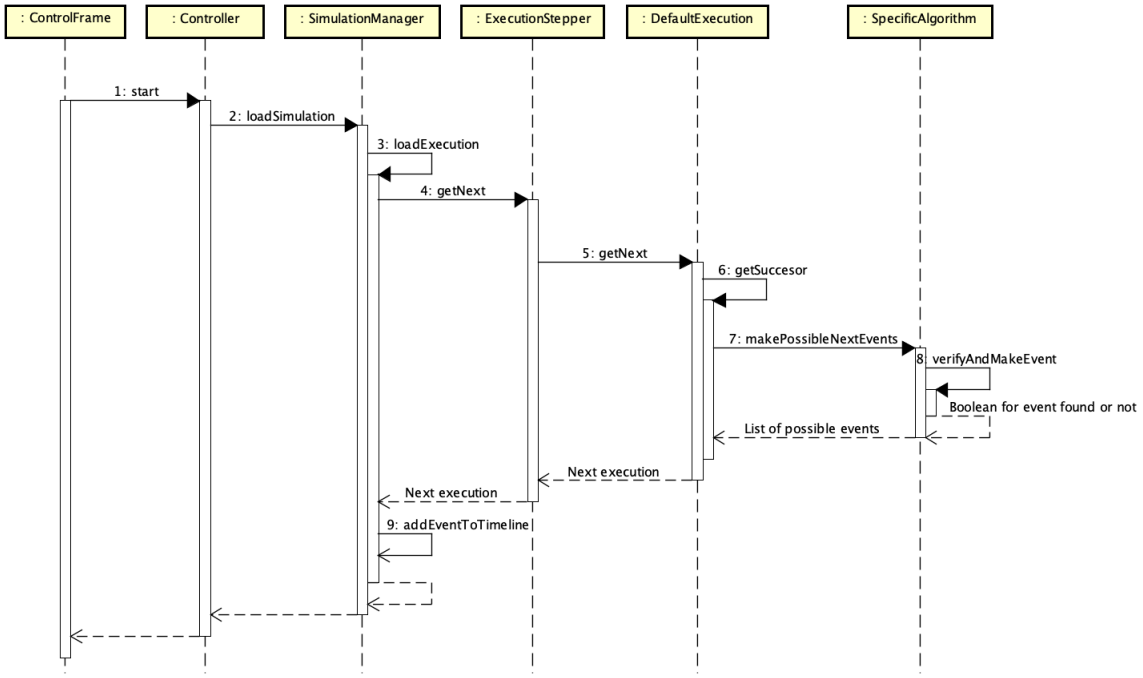


Fig. 8: Main execution of the simulation in the Java port.

Tree and Echo algorithm was validated, the Tarry implementation in native Java was validated by comparing with the reference book [4, p. 20-21]. Note that the resulting computation, although valid, was not identical to the computation described in the book, since there is a number of valid computations for a single algorithm in a given network topology. For instance, it is a design choice of DaViz to always select the first process in its neighbouring list to forward or send a message. This simplifies the implementation but reduces the number of possible valid computations that are simulated in DaViz. Additional evaluation was done by running the simulation in different network topologies, varying the number of processes in the network and the channels between processes. Once all evaluated computations returns a valid execution, Tarry is set as ready to use in the Java port of the simulation.

Since the prototype infrastructure developed in the Java port is now validated, more algorithms will be added in future stages of the project.

## V. Software Package

DaViz is publicly available in a GitHub repository[17]. The executable JAR distribution of DaViz and the source code in that specific version are available in the repository releases page.

## VI. Related Works

In this section, we present some similar tools developed for the visualization or simulation of distributed algorithms, describing their aspects and execution. Table II summarizes the main aspects of these tools.

### A. ViSiDiA

ViSiDiA, first introduced in 2001 [12] and later redesigned in 2014 [7], is a Java Framework for the design, **Vi**sualization and **Si**mulation of **Di**stributed **A**lgorithms. It was developed to support researches and assist in the learning process. It provides a vast range of distributed algorithms, synchronous or asynchronous executions, mobile agents and sensors networks. It allows users to implement their algorithm using the Java programming language. Alternatively, users can define a distributed algorithm by simply specifying rewriting rules.

The system is designed using a multi-layer architecture, where components are as independent as possible from each other. First, the GUI enables to build the network topology and to visualize the progress of the simulation in real-time. The second layer has two components: the network description and the simulation module. The simulation interprets events and updates the network. Due to its event-driven nature, ViSiDiA does not depend on the GUI for execution. The last layer consists of an API for the algorithm library, which allows users to define their algorithm.

### B. Distal

Distal [8], introduced in 2013, is a Scala[18] framework for implementing fault-tolerant **Dist**ributed **al**gorithms. It proposes a simple language for defining algorithms, converting pseudocode to efficient executable code. It consists of two parts: a domain-specific language (DSL) based in state machines, in which algorithms are described, and a message layer that addresses low-level issues, such as connection

[17]https://github.com/praalhans/DaViz

[18]https://www.scala-lang.org/

| Paper Info | | Tool | | | | Has Module | | User Defined Algorithm | | Runs on |
|---|---|---|---|---|---|---|---|---|---|---|
| Ref. | Year | Name | Language | For Research | For Learning | Simulation | Vizualization | Has? | Language | |
| [2] | 2017 | DaViz | Java<br>Frege | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ | Client-only |
| [7] | 2014 | ViSiDiA | Java | ✓ | ✓ | ✓ | ✓ | ✓ | Java<br>Rewriting Rules | Client-only |
| [8] | 2013 | Distal | Scala | ✓ | ✗ | ✓ | ✗ | ✓ | DSL | Client-only |
| [9] | 2006 | LYDIAN | TCL/TK | ✗ | ✓ | ✓ | ✓ | ✓ | C based | Client-only |
| [10] | 2004 | DAP | C++ | ✓ | ✓ | ✓ | ✓ | ✓ | Any | Client–Server |
| [11] | 1998 | VADE | Java | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ | Client–Server |

TABLE II: Comparison between some related tools.

management, threading and (de)serialization. The usage of DSL allows Distal to be simple and intuitive by abstracting the implementation details. Both components are designed independently and allow to easily replace the message layer for another communication system.

### C. LYDIAN

LYDIAN [9], introduced in 2006, is a **L**ibra**y** of **Di**stributed **A**lgorithm and A**n**imations. It was developed to support the teaching and learning of distributed algorithms. Users can define an arbitrary network topology and select algorithms and animations for simulations. Alternatively, users can define their custom algorithm description using a high-level language based on the C syntax, simulate it and visualize the resulting execution. The LYDIAN simulator module uses the concept of communicating finite state machines and is event-driven and the GUI is implemented with TCL/TK[19], a Tool Command Language for dynamic programming.

### D. DAP

DAP [10], introduced in 2004, is a **D**istributed **A**lgorithms **P**latform. It provides a generic and homogeneous simulation environment, allowing the implementation, simulation and testing of distributed algorithms. DAP is implemented in C++[20] and allows users to implement and run custom protocols using any language, as long as the specified communication protocol is followed. The user can monitor and control the algorithm execution through the tool GUI.

### E. VADE

VADE [11], introduced in 1998, is a tool for the **V**isualization of **A**lgorithms in **D**istributed **E**nvironments. It allows simulating and visualizing the asynchronous execution of distributed algorithms. It is a client-server architecture, where simulations of algorithms are executed in the server and the results are sent to the client for visualization. It is implemented using the Java programming language, using processes in the server and applets in the client.

[19]https://www.tcl.tk/

[20]http://www.cplusplus.com/

### F. Others

There are many other tools developed to simulate (and visualize) distributed algorithms. Some additional examples are Zada [13], IOA [14], VEDA [15], DAJ [16], Khanvilkar [17] Mace [18], PeerSim [19]. However, it is not the goal of this report to perform a detailed state-of-the art analysis on such tools.

## VII. Conclusions

This report describes improvements implemented in the DaViz project. The majority of the improvements targets developers and has the goal to improve the development process, environment and collaboration. Improvements for the end-user include a lightweight port of DaViz and ensuring the possibility of running simulations with the Tree and Echo algorithms.

### A. Future Works

*1) Automated validation of the correctness of Frege and Java ports:* The Java implementation of the Tarry algorithm has no automated validation at the moment. Its validity was manually verified against the reference book and the Frege port. It would be useful to have an automated comparator class that would validate the results of both ports upon certain events (*e.g.,* pushing commits to the repository) or when manually triggered. There are two possibilities considered for this validation.

The first is implementing a logger and comparator classes using the console as UI. The logger would listen to the result of the simulation and store it as a JSON file. The comparator would take the resulting JSON of both ports and verify if the results are equal. For simplicity, built-in tools from the OS console, such as the `diff` command[21], could be used as a comparator.

The second is leveraging Java automated tests to assert the result of both ports. Since the classes within the ports share the same namespace (package), they would conflict if directly compared to each other. Instead, a possible approach is to define the network and the expected result in the test. Each port is individually compared to the expected result. If both

[21]http://manpages.ubuntu.com/manpages/eoan/man1/diff.1.html

succeed, then we can infer that they are correct, otherwise, mismatches occurred.

To address non-deterministic choices, one possible approach would be defining an enumerated set of choices that can be taken in a given step of the simulation. Each number in the sequence refers to an option available in the Choice window. Deterministic runs would always result in the first index, undefined indexes would indicate an error in the execution, and any other index would indicate a non-deterministic step. A choice sequence is pre-defined and used to simulate an algorithm in both ports. Their results are then compared and should be the same, given the simulation was driven by the provided choice sequence.

There are, however, edge cases that bring additional complications, such as the possibility of infinite runs and dead-lock.

*2) Design and implement a control algorithm on top of a basic algorithm:* Control algorithms are more complex to simulate than basic/computation algorithms since we need to simulate the execution of two algorithms. In the design of DaViz, there is an event loop where the simulation requests the next execution. In this request, algorithms evaluate the current state, verify what are the possible next choices, generate events for these choices and update their process states with one of the created events. The control algorithm would act as a wrapper around the state at a single process, which provides a method for generating events. To produce snapshots of the local state within a process and attempt to generate events in the future, the algorithm object should be clonable.

When basic algorithms are created, each process is informed of its neighbouring processes. When a control algorithm is created, it should receive information about the basic algorithm. Since both algorithms implement the same base interface, both have an internal event generator implementation. Thus, the control algorithm can invoke the event generator of the basic algorithm. Afterwards, the control algorithm can take an appropriate action based on the event created. For internal events, the control algorithm would simply create its internal event. For send events, the control algorithm can piggyback the event by wrapping the message with an additional payload. For receive events, the control algorithm would create a receive event that receives the wrapped message and passes the original message (if any) to the receiver process. When the processes send and receive messages, the control algorithm can decide to set their states to inactive.

*3) Implement more basic algorithms:* An obvious extension of this project is to implement more basic algorithms. With the introduction of the native Java port, there is an option to add more algorithms in Java or Frege. When implementing an algorithm not supported yet in any port, it is important to implement the common behaviour in the Core project.

The next step in the Java port is implementing the wave algorithms already available in the Frege port. Taking the reference book, the next set of algorithms suitable to be implemented in Frege are routing algorithms. One could start with Chandy-Misra [4, p. 53-55], which assumes a centralized and undirected network. Then proceed to Merlin-Segall [4, p. 55-28], a variation of Chandy-Misra where the shortest path is calculated in rounds. The reference book describes two more routing algorithms.

Some adjustments are required to implement routing algorithms. For a starter, new assumptions need to be considered: undirected and weighted networks. While the infrastructure of DaViz already supports a certain level of undirected networks and giving weights to channels, it is not enabled, completed or tested.

*4) More flexibility in selecting a neighbour process to send messages:* Currently, DaViz takes a sequential approach to decide the next process to send a message to. It simply takes the next process from its neighbour list. To offer more possible executions of an algorithm in a given network, it might be interesting to allow selecting any of the neighbour processes.

This could be accomplished by generating a possible next event for each of the neighbour processes. Since the default execution always takes the first possible choice, it should still be possible to predeterminate the execution sequence.

*5) Fixing the Choice windows in Java port:* The introduced Java port of DaViz described in this report currently has a bug: no option is available in the Choice window or, if available, it throws an exception when selecting it. To provide the user with the same experience in both ports, the Choice window should be usable in the Java port as well.

*6) Allowing user-defined algorithms:* Similar to some of the tools described in section VI, it would be interesting to add a feature to DaViz allowing users to define their custom algorithm for simulation. The current design of DaViz already provides some support to provide this feature. In addition to its event-driven design, the algorithm assumptions, events and information within an event are thoroughly defined in DaViz. Ideally, user-defined algorithms would simply be a set of configurations and/or rules. It should consist of a set of assumptions, mapping of current states (*e.g.,* has the token, neighbour list, parent process, etc.) to a given event (*e.g.,* send, receive, internal, result) and the processes initial state.

*7) Minor improvements in the UI and general features:* There is a series of minor improvements that can be done in DaViz. Providing a feature to save and load networks and even an execution would be useful. Whenever DaViz is closed, the user must specify the network again, which can be a tedious task. Another benefit in this feature would be the possibility of sharing the network or a given computation with other students or to discuss with the lecturer. To store the network, a simple configuration of nodes and channels would be sufficient. For the execution, given that the choice sequence described in VII-A1 is implemented, it could be leveraged to store this sequence.

Besides allowing to save the network, it would be interesting to already provide default networks and computation scenarios for each algorithm. Given the examples described in the reference book, it should be simple to map them as examples within DaViz.

Another improvement is redesigning the UI to use a single main window (frame) with the current five windows as multi-

ple sub-windows, similar to how IDEs are designed. There are two reasons for opting for a single main window. First, it is simply more intuitive for the user. Furthermore, when closing any the windows, even the optional Information and Choice windows, the entire application is closed. Second, a side effect of multiple windows when running DaViz in debug mode is that four windows stay active and unresponsive when hitting a breakpoint. Although the windows can be moved around, it is a tedious and unnecessary task.

Other improvements in user experience are: undo/redo features, synchronizing the Timeline window with the Network window (*e.g.,* displaying the token being sent from a process $P$ to a process $Q$), auto-selecting a random process as the initiator, auto-generating random networks given an algorithm assumption, a minimum and maximum number of processes and connections between processes.

## VIII. Acknowledgements

I would like to thank Hans-Dieter Hiep and professor Wan Fokkink. Hans-Dieter, who originally designed and implemented DaViz, for all the support he provided in explaining the tool features, structure and implementation design, as well as participating in meetings to discuss the next improvements. Wan for actively being involved in the project and the meetings to discuss the next steps. And especially for being understandable when the proposer was having a difficult period in his academic and professional life.

## References

[1] G. McCluskey, "Using java reflection," Jan 1998, retrieved on 15 March, 2020. [Online]. Available: https://www.oracle.com/technical-resources/articles/java/javareflection.html

[2] H.-D. A. Hiep, "Technical report: Daviz," http://www.hansdieterhiep.nl/2017/05/daviz.html, Vrije Universiteit Amsterdam, Department of Computer Science, Tech. Rep., May 2017.

[3] V. Driessen, "A successful git branching model," https://nvie.com/posts/a-successful-git-branching-model/, January 2010, retrieved on 1 March, 2020.

[4] W. Fokkink, *Distributed Algorithms – An Intuitive Approach*, 1st ed. Cambridge, Massachusetts London, England: The MIT Press, 2013.

[11] Y. Moses, Z. Polunsky, A. Tal, and L. Ulitsky, "Algorithm visualization for distributed environments," 11 1998, pp. 71 – 78, 154.

[5] B. Ellis, J. Stylos, and B. Myers, "The factory pattern in api design: A usability evaluation," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 302–312.

[6] A. Shvets, "Abstract factory," https://refactoring.guru/design-patterns/abstract-factory, 2019, retrieved on 1 February, 2020.

[7] W. Abdou, N. O. Abdallah, and M. Mosbah, "Visidia: A java framework for designing, simulating, and visualizing distributed algorithms," in *2014 IEEE/ACM 18th International Symposium on Distributed Simulation and Real Time Applications*. IEEE, 2014, pp. 43–46.

[8] M. Biely, P. Delgado, Z. Milosevic, and A. Schiper, "Distal: A framework for implementing fault-tolerant distributed algorithms," in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2013, pp. 1–8.

[9] B. Koldehofe, M. Papatriantafilou, and P. Tsigas, "Lydian: An extensible educational animation environment for distributed algorithms," *ACM Journal on Educational Resources in Computing*, vol. 6, 06 2006.

[10] I. Chatzigiannakis, A. Kinalis, A. Poulakidas, G. Prasinos, and C. Zaroliagis, "Dap: a generic platform for the simulation of distributed algorithms," in *37th Annual Simulation Symposium, 2004. Proceedings.*, April 2004, pp. 167–177.

[12] M. Bauderon, S. Gruner, Y. Métivier, M. Mosbah, and A. Sellami, "Visualization of distributed algorithms based on graph relabelling systems," *Electronic Notes in Theoretical Computer Science*, vol. 50, no. 3, pp. 227 – 237, 2001, gT-VMT 2001, Graph Transformation and Visual Modeling Techniques (Satellite Workshop of ICALP 2001). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066104001744

[13] A. Mester, P. Herrmann, D. Jager, V. Mattick, M. Sensken, R. Kukasch, A. Ritter, S. Bunemann, P. Unflath, M. Bernhard *et al.*, "Zada: Zeus-based animations of distributed algorithms and communication protocols," 1994.

[14] S. J. Garland, N. A. Lynch, and M. Vaziri, "Ioa: A language for specifying, programming, and validating distributed systems. draft," *Unpublished manuscript, September*, 1997.

[15] C. Jard, J.-F. Monin, and R. Groz, "Development of veda, a prototyping tool for distributed algorithms," *IEEE Transactions on Software Engineering*, vol. 14, no. 3, pp. 339–352, 1988.

[16] M. Ben-Ari, "Interactive execution of distributed algorithms," *ACM Journal of Educational Resources in Computing*, vol. 1, p. 2, 08 2001.

[17] S. Khanvilkar and S. M. Shatz, "Tool integration for flexible simulation of distributed algorithms," *Software: Practice and Experience*, vol. 31, no. 14, pp. 1363–1380, 2001.

[18] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat, "Mace: language support for building distributed systems," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 179–188, 2007.

[19] A. Montresor and M. Jelasity, "Peersim: A scalable p2p simulator," 09 2009, pp. 99–100.