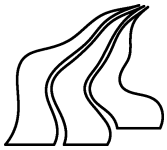
The TraceInvader

Mikkel Christiansen Jesper Langfeldt Hagen Kristian Qvistgaard Skov

January 14, 1997

Aalborg University
INSTITUTE 8 - Institute for Electronic Systems

Department of Computer Science



Title:
TraceInvader—A tool for debugging distributed applications.

Theme:
Debugging distributed applications.

Time period:
01.09.96–07.01.97

Semester:
DAT5/F9D

Project group:
E1-104b

Participants:
Mikkel Christiansen
Jesper Langfeldt Hagen
Kristian Qvistgaard Skov

Supervisor:
Arne Skou

Number of prints: 9

Pages: 59

Appendix pages: 10

Abstract:
This project deals with the discipline of detecting bugs in distributed applications. The general problems concerning bugs in distributed applications are identified and requirements for a trace-based debugging tool are stated. Based on this a theory for observing and visualizing behavior is developed and a prototype tool is implemented. Included in theory are algorithms for establishing partial orders between events from a distributed application which causes minimal perturbation on application behavior. To reduce the complexity of debugging distributed applications the use of causal graphs is argued.
The prototype implementation includes generic features for generating traces from any distributed application using the message passing paradigm.
The results from the prototype debugger concludes that combining theories for observation of distributed application with methods for visualizing behavior can solve the observation problem caused by the lack of global timing and reduce the complexity for debugging distributed applications.

Preface

This report represents the first part of a masters thesis written within the Distributed Systems group at the Department of Computer Science at Aalborg University. The report is written in the period 1/9 1996–7/1 1997 by group E1-104b.

The foundation for this masters thesis was an abstract—[CHS96]—developed as the result of a study of issues related to detecting of bugs in distributed applications.

In the abstract the decision was made for the development of a debugging tool for distributed applications. The goal of this project is to design and implement such a tool.

In the report references are made to various literature, which can be found in the bibliography. The syntax for references is [Lam78] for a reference to an article by *Leslie Lamport* written in 1978.

Aalborg University, January 7, 1997

Mikkel Christiansen

Jesper Langfeldt Hagen

Kristian Qvistgaard Skov

Contents

Chapter 1

Introduction

In the last few years the use of computer networks has increased dramatically. But we have only just seen the beginning of network based information systems. These technologies will no doubt cause significant changes in the years to come. As the dependency of network based systems increases, so does the need for methods for developing reliable and efficient applications using network communication. This project deals with the notorious discipline of finding errors in such software; debugging distributed applications.

This first chapter begins with definitions related to *distributed applications* and examines the special problems related to the development and debugging of these. Having described the fundamental issues related to the debugging of distributed applications, we then analyse present technologies. After we have described and identified the limitation and weakness of present technologies, we propose ideas and requirements for a new debugging tool. This chapter finishes with a description of the layout of the report.

1.1 Definitions

Since the terms used in the areas of distributed applications and the debugging of such vary throughout the literature, we will come up with an informal framework covering the basics in the following.

1.1.1 Distributed applications

A distributed application is a program consisting of two or more communicating tasks running in a MIMD environment. The term MIMD (Multiple Instruction stream, Multiple Data stream; see [Tan95]) includes any type of architecture with one or more independent computers each with independent program counters, program and data. This could be workstations in a LAN, a hyper-cube or a transputer.

The definition of distributed applications makes no assumptions about the a specific communication paradigm. In principle this could be message passing as well as shared memory or other paradigms. However, when dealing with network based applications, the message passing paradigm is the most prominent. In [Ray88] a distributed algorithm (which can be generalized to a distributed application) is characterized by the relation:

$$\text{distributed algorithm} = \text{processes} + \text{messages} \tag{1.1}$$

This relation supports the notion of a distributed application using message passing.

In [LM89] a distinction is made between parallel and distributed applications based on how tightly the modules of the applications are coupled. However when generally dealing with the task of finding bugs, the degree of coupling between modules is irrelevant.

In this report we restrict our focus to debugging applications using message passing for communication.

1.1.2 Tasks, events and traces

The executing entity in a distributed application is called a *task*. A task is an implementation of a sequential algorithm, and is thus a sequence of mutual exclusive actions or *events*. An event could for example be a simple statement, the creation or termination of a task, or a communication primitive.

A *trace* is a recording of an execution, that is, a sequence of events actually occurring in the application during the execution. Traces are generated by observing events of interest occurring in the application. Observation of events can be done either inside or outside the application. When observing from within the application, each event results in a notification of the observer. In outside observation, the observer typically has full control and knowledge of the application.

1.1.3 Communication

In this report we focus on distributed applications communicating through message passing. We will distinguish between two types of communications:

Synchronous communication This corresponds to a synchronization between the participating tasks. The events corresponding to the synchronization primitives are conceptually happening synchronously, that is the message communicated is sent and received at the same time in all participating tasks.

Asynchronous communication The sending event happens before and is mutual exclusive with the receiving events. That is, time passes with the message.

Under the distributed application lies the *message passing system*—or *MPS* for short. MPS's comes in many flavors. Despite this, each MPS will have some basic properties. The goal of a MPS is to provide the programmer with primitives for delivering messages between tasks. These messages can be communicated in many ways. The most common is a peer-to-peer communication, which is a communication of a message from one task to another. Group communication is used when more than two tasks participate in a more advanced communication—the simplest group communication is broadcast communication.

1.1.4 Validation of applications

The validation of an application is divided into three activities:

Verification A verification is a check of whether or not an algorithm fulfills an abstract specification. In tools like Spin (see [Hol91]) the algorithm is implemented in a formal language and the specification is implemented by means of boolean expressions that must hold at specified states in the algorithm.

Test An algorithm can be seen as a function mapping input to output. When testing, a specification of this function is compared with the actual function computed by the algorithm. One distinguishes between black-box and white-box testing. In white-box testing, the programmer applies knowledge of the internal workings of the implementation, when she designs the test. The black-box test assumes no such knowledge.

Debugging When an algorithm has been implemented and an unforeseen error occurs, the debugging activity can be used to help locate the error. Typically the algorithm is run within a debugging tool and the programmer has full control over the execution of the algorithm and can see the path of execution. This type of debugging is called runtime debugging. In postmortem debugging the path of execution is recorded in a trace. The programmer can then see the path of execution without having control of the execution.

Verification is typically used in the design phase of the development cycle, while test and debugging is used during or after the implementation phase. Thus the debugging tool we are designing and implementing in this project, is a tool to be used during or after the implementation phase.

1.1.5 Bugs in distributed applications

In [BHD+95] bugs in distributed applications are divided into logical bugs and performance bugs:

Logical bugs These are bugs violating the expected logic of an application. A simple example could be a missing reply from a task in a client/server application. Whether the bug is caused by design flaws or errors in the implementation, makes no difference—it is a logical bug in both cases.

Performance bugs These are bugs violating the performance of an application. A typical performance bug is a performance bottle-neck, i.e. the input and output rate of a part of the application are too far apart.

In this report we focus on the location of logical bugs in distributed applications. Thus, our debugging tool will not directly support the task of locating performance bugs. The main reason for this decision is the large base of existing debuggers for performance analysis and the relative small base of debuggers for locating logical bugs.

1.2 Debugging distributed applications

As the nature of distributed applications differs from sequential applications, so does the issues related to debugging distributed applications differ from debugging sequential applications.

Sequential programs are characterized by the flow of control always running in the same direction. When debugging a sequential application the execution can be observed in atomic steps. In theory, the whole state of the application is always available between each single step of execution. Given the same input, a sequential application will exhibit the same pattern of execution—a sequential application is deterministic. This enables the debugging activity to be based on several executions showing the same behavior.

Debugging distributed applications is a more complicated activity. Applying standard methods for sequential debugging is not suitable. In fact applying the usual methods for distributed applications will in many cases be misleading. Trying to capture the exact behavior of a distributed application can be a frustrating task.

Due to the parallelism in distributed applications, the debugging of such suffers from problems not present in the debugging of sequential applications. The most central of these problems are:

Complexity As mentioned earlier, a distributed application can be seen as a collection of several communicating sequential tasks. It is simple to get an overview of a single sequential task, since it is simply a sequence of events. It is more complicated to gain an overview of several communicating sequential tasks. In the distributed case, a bug in one task can disturb the behavior of another task. This also means, that it can be difficult to find the origin of a bug, since it can be caused by another bug happening earlier in another task.

Non-determinism A distributed application with more than one task can have several possible paths of execution on which a programmer has no influence. This non-determinism in distributed applications can be caused by varying propagation delays and non-deterministic scheduling of tasks. A side-effect of having non-determinism is called the *state-explosion* problem, which means that at each point of non-determinism, several future paths of executions exists. This means that the number of possible states explodes.

Reproducibility of executions is the ability to execute an application several times and observing the same behavior. Having reproducibility in debugging context eases the process of debugging. Non-determinism excludes reproducibility of executions, which means that methods for removing or controlling non-determinism in distributed applications must be provided, if reproducibility is to be supported in a debugging tool.

Probe-effect When a distributed application is observed, there is a danger of a *probe-effect*. A probe-effect is the disturbance of the execution due to the observation; therefore it is also commonly called *perturbation*. The probe-effect is said to have no importance, if it does not create or hide bugs in the application. If these bugs are of the logical kind, the probe-effect must not alter the intended path of execution and communication pattern of the application, and if they are of the performance kind, the probe-effect must not disturb the intended performance.

Observation in most cases causes a probe-effect, and the probe-effect is often proportional to the amount of information gathered in the observation. This reflects a dilemma, since the probe-effect can distort the execution of the application, while as much information as possible is often needed in order to provide a good foundation for locating bugs.

Global timing In order to determine the path of execution of a distributed application, events observed during the execution is often stamped with a global time. In sequential applications, having a concept of global time, this poses no problem, and it is easy to establish the path of execution by merely using time-stamped events. In distributed applications there is typically no concept of global time, since each processor used in the application has its own clock, which is not synchronized with a global clock. In order to establish the actual ordering of events by using time-stamps, it is necessary to provide time-stamps that solve these problems.

Because of these problems, traditional approaches to debugging sequential applications does not apply directly to debugging distributed applications. The area of debugging distributed applications calls for other methods. In the next section, we will examine the methods currently used by existing debuggers for distributed applications for handling these problems.

1.3 Present technologies

In this section we will analyse present technologies for debugging distributed applications. Through a description of different approaches for handling the problems mentioned above, we will describe the

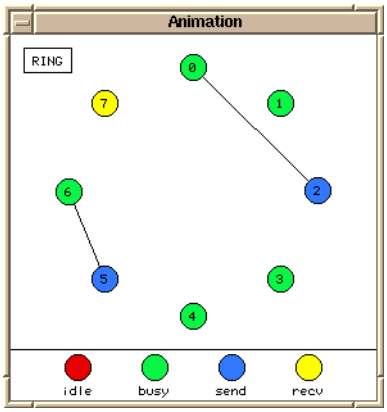


Figure 1.1: This is a snapshot visualization from ParaGraph. Each circle represents a processor, and the color of the circle indicates it's status. Lines between circles indicates a communication between processors.

main areas of debugging techniques for distributed applications.

1.3.1 Complexity

Methods for handling the complexity of distributed applications can be separated into two main areas—methods for visualizing application behavior and formal methods.

Visualization

There are mainly two benefits to be achieved from the use of visualizations. A graphical presentation of a trace can help reduce the complexity of large sets of trace data. In addition to this, visualizations can help reflect the distributed nature of applications and the specific algorithmic characteristics of applications.

The majority of tools for visualizing the behavior of a distributed application are based on a traces of events. An example of a tool for visualising the behavior of an application is ParaGraph which is described in [HE91, HMR95a, HMR95b, TU91].

We distinguish between three types of visualizations. These are

- snapshot visualizations,
- historical visualizations, and
- cumulative visualizations.

A snapshot visualization shows the state of the distributed application at a particular moment in time. Therefore this type of visualization is also called an animation visualization—or animation for short. Snapshot visualizations are useful for showing the transitions from one state to another—thereby revealing the behavior of the application. A typical snapshot visualization shows status information on tasks or communication. In figure 1.1 is shown a typical snapshot visualization.

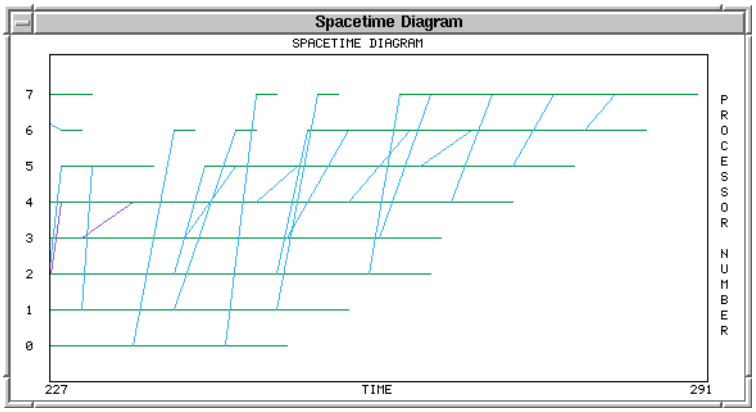


Figure 1.2: This space-time diagram is taken from ParaGraph. Each horizontal line represents the time-line of a processor. Each line drawn from one time-line to another represents a communication between processors.

Historical visualizations shows the history of a computation instead of just showing the present state. This is useful when the programmer is interested in looking in patterns in the behavior. It is possible to see behavioral patterns in a snapshot visualization, but it is easier to compare patterns directly when looking at a full history of past states. In figure 1.2 is shown one of the most used historical visualizations—namely the space-time diagram.

In some situations, bugs are revealed not through the behavioral patterns but in the overall result of the computation. In situations like these, summaries or statistics of the whole computation are more feasible. These can be visualized in a cumulative visualization. Typical cumulative visualizations shows the total number of messages sent, received or lost, or the processors utilization or idle time. An example of a cumulative visualizations is seen in figure 1.3.

In order to ensure that a visualization actually reduces the complexity of the debugging process, it is necessary to carefully plan the means used for creating the visualization. Many of the visualizations we have seen in existing debuggers, overflows the user with large overwhelming visualizations with more information than the user can comprehend. One of the best tools for avoiding this, is the use of abstraction in the visualization. The abstractions used can be based on the structure of the algorithm being debugged, patterns in the communication, or perhaps the underlying topology of the MPS.

Formal methods

Other tools like EBBA (Event Based Behavioral Abstraction; see [Bat95]) deals with complexity through a more formal approach. This tool differs from formal verification tools because it performs model checking on traces of actual executions of the implementation and not just abstract models of the implementation.

Rather than dealing with the complexity of the complete state space, tools like EBBA enables the user to focus on the actual application behavior. This approach uses traces from an application and compares these traces with models specified in a form of regular expressions. Other systems like the debugger developed for the Amoeba distributed operating system (see [Tan95]) uses the same approach. Complexity of program behavior is reduced through mechanisms for *filtering* traces of events and model checking is implemented through *recognizers*. These are also specified in a form of regular expressions. In [CM91]

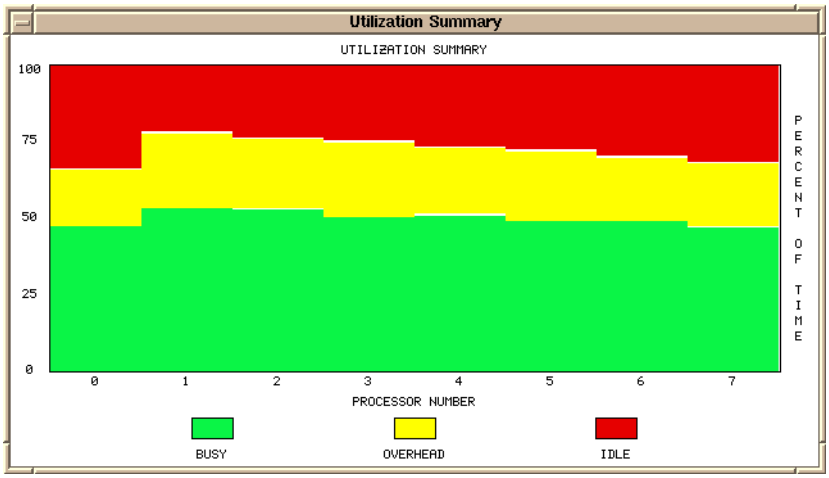


Figure 1.3: This cumulative visualization from ParaGraph shows the utilization of the processors.

is presented methods for checking temporal predicates during the execution of an application.

Common to most formal methods is that some sort of specification in a formal language is necessary. This means, that the programmer has to use time and energy in designing and implementing this specification. We are of the opinion that the ordinary programmer want a more soft approach to debugging—an approach that can be learned and used within 5 minutes, that is, the debugger must be ready-at-hand.

Other methods

Methods for visualization and formal methods are the main methods for handling complexity. However, other methods exist that provides different mechanisms for abstraction.

An often used debugging method is static debugging, i.e. the insertion of output statements in the distributed application. This can be viewed as a method for reducing complexity. Output statements are inserted selectively and the text output is appropriately chosen. This means that it is easy to compare the generated output with the expected output. The drawback with this method, is the static nature of the method (hence the name.) Since statements are inserted directly in the source code, they tend to clutter it. Re-compilation is also necessary whenever the output statements are changed, added or deleted. Because of this, statements are often inserted less selective than intended, thus complicating the generated output.

[KB95] describes an alternative approach to debugging, which is a kind of reverse engineering. The author argues for mechanisms for automatically generating an architectural description of a distributed application. These mechanisms makes clusters of tasks having identical properties using statistical approaches. Through visualizations, the application can then be viewed at different levels of abstraction, and subsystem structures are shown. This exposure of the actual architecture of the application can then be compared to the designed architecture, thus revealing differences, that could be caused by bugs.

Probably the most exotic method for reducing complexity, is the use of sound. This is called auralization. In an auralization, the behavior of a distributed application is mapped to sounds. The idea is to provide a mapping that reveals bugs—i.e. that sounds out abnormalities in the behavior. Work in this area has been presented in [FAJ91, RAM+92, Tav94]. Like visualizations, auralizations has the ability to convey

a large amount of information. However, auralizations do not scale very well—they tend to be confusing in large distributed applications with many concurrent events, thus producing many overlapping sounds. It has been proved, though, that the combination of auralization and visualization is powerful.

1.3.2 Non-determinacy

The problems of non-determinacy emerges as a result of communicating tasks running concurrently. The main approaches for handling nondeterminacy use methods for detecting nondeterminacy. These are divided into three categories:

Compile time analysis Nondeterminacy is detected by making use of program semantics. This is independent of input data.

Post mortem trace analysis Nondeterminacy is detected through trace analysis.

On-the-fly methods During execution, information is collected and nondeterminacy is then detected.

[DKF93] combines compile time analysis and on-the-fly methods for the detection of nondeterminacy. All nondeterminacies are distinguished to be intended or unintended (caused by a bug). The detected nondeterminacy is then used in *controlled execution*, where the application is executed and suspended at each point of nondeterminacy. A similar approach is seen in [EP89] which also classify different levels of non-determinacy such as *conclusive non-determinacy*, which is independent of input data, and *input dependent non-determinacy*. Both compile time analysis and post mortem trace analysis is used in this approach.

When the nondeterminacies of an application is known, it is possible to use this to systematically check all paths of execution and reproduce an execution. Also, bugs causing nondeterminacy, are easily detected, by examining the detected nondeterminacies.

1.3.3 Probe-effect

The standard method of observing the behavior of distributed applications is through the insertion of event generating statements in the application. This is called *instrumentation*. The probe effect can occur as a result of this instrumentation. Several ideas exist for reducing the disturbance of this instrumentation.

Basically there are two types of instrumentation: *static instrumentation* and *dynamic instrumentation*. Reed et al. describes in [RAM+92] selective static source level instrumentation to reduce the probe effect. A similar method is described in [Jak95]. An alternative to the source level instrumentation techniques is described in [MCC+95]. Instead of altering the source code, instrumentation statements are inserted in the binary code. This removes the necessity to recompile the application each time new events are to be generated—such a necessity can cause programmers to insert more events than necessary in order to minimize the compilation time.

A different approach proposed by Rubin et al. [RRZ89] uses hardware functionality instead of instrumentation. The idea is for each processor to have a mirror processor for monitoring the behavior of the application. This is combined with filtering in order to reduce the size of the trace-file. Even though the idea dramatically can reduce the probe effect, it is an expensive alternative to instrumentation.

1.3.4 Global timing

Several attempts has been made at making algorithms for synchronizing all clocks in a network with a global clock. One of these is the *Network Time Protocol* (NTP; see [Mil91]). Others are described in [CDK94]. Common to these are the low precision of the so-called synchronization. NTP has been measured to have an error in the synchronization that is below 30 milliseconds. Still, this is not sufficient, since millions of processor instructions can be executed in a time interval of this length.

Instead of using real-time clocks, ordering events in the application can be done by introducing a central server to which all events are sent. In this way an ordering can be posed on the events which reflects reception. Basically there are two problems with this approach, mentioned in [Car96]. First of all it takes time to send the events to the server. With varying propagation delays, this can cause events to be received in an order not corresponding to the actual order. Secondly, this can cause a probe-effect since the sending process will be slowed down.

In 1978 Lamport proposed in [Lam78] a solution for the problems related to the ordering of events using *logical clocks*, which are an approximation to global time. He defined a relation *happened before* from which relations between events can be analysed. The happened before relation introduced a partial ordering of events and an implementation of this relation was used by Lamport to create a total ordering for a process scheduling algorithm. Several people has proposed refinements of Lamport's idea of logical clocks and partial ordering of events. Among these are [Fid91, Mat89]. Currently, the use of logical clocks seems to be the most feasible solution to the global timing problem. Despite this, the use of these theories for observing distributed applications has been limited.

1.4 The goal of this project

The purpose of this project, as mentioned earlier, is the design and implementation of a debugging tool for debugging distributed applications for logical bugs. In this chapter, we have mentioned the central problems in the area of debugging distributed applications, and the finest goal would be to solve all of these. Despite the many tools we have examined, some of which we have mentioned in this chapter, we have not found a tool trying to solve all of the problems in one go. Each tool tends to be focused at trying to solve a small subset of these problems.

Based on experiences with tools such as ParaGraph, the use of visualization was revealed as a powerful mechanism for dealing with complexity. Since logical bugs are exposed by observing application behaviour, it is in our interest to ensure that the observed behavior corresponds exactly to the actual behavior. This means that both the problems of global timing and probe-effect must be considered, since the presence of these problems can obscure the observed behavior. The most promising method for handling the global timing problem was the use of logical time. These observations leads us to the following hypothesis:

> In order to locate logical bugs in a distributed application, the following is needed:
>
> - Observation of the behavior of the application must be reliable, i.e. true to the actual behavior. By handling the global timing problem with the use of logical time, and by ensuring minimal probe-effect, this reliability is maximized.
> - It must be easy to gain an overview of the application behavior. Hence, the complexity problem must be handled. This can be done through the use of visualizations.

In the following we will describe further requirements to the debugging tool, that will be the result of this project.

1.4.1 Requirements to a debugging tool

In the construction of a debugging tool which can help us making conclusion on the abovementioned hypothesises, the tool must be based on:

- *The requirements from the hypothesis*. The complexity problem is handled through visualizations, the global timing problem is handled through the use of logical time, and the probe-effect is minimized.
- *Postmortem trace analysis*. By the use of instrumentation in order to generate events and collecting these in a trace, the probe-effect can be minimized. The debugging tool will be based upon traces.
- *Methods for ensuring the generality of the tool*. We do not wish to limit the functionality of the tool to a specific MPS. Therefore the debugger must support methods for generating traces from any type of MPS.
- *Support for several visualizations*. As mentioned in section 1.3.1, there are different types of visualizations, each designed to reveal different types of bugs. Therefore, one visualization is not necessary.
- *Source level debugging*. Experiences gained from working with debuggers have revealed a simple but very important need. When trying to locate a bug by examining traces of events, a relation between source code and events is essential. This enables the programmer to see exactly what line of code that is responsible for the generation of an event.
- *Support for static debugging*. This we require, because static debugging is still one of the most used debugging methods.
- *Ready-at-hand*. In order to ease the use of the debugging tool in the process of debugging, the tool must be *ready-at-hand*, which means that it must be user-friendly and it must be easy to learn the use of it.

1.4.2 Outline of the report

Chapter 2 This chapter describes existing methods for ensuring reliable observation of application behaviour.

Chapter 3 Based upon the observations in the previous chapter, new methods are developed that offers a better support for the debugging activity. A visualization which has the goal of handling the complexity problem will be developed, and it's functionality described.

Chapter 4 The architecture of the modules in the debugging tool is described here. Each module is also described superficially, leaving a thorough discussion to the later chapters.

Chapter 5 The module, which handles the storage of traces is described here.

Chapter 6 All observation of distributed applications and generation of traces is handled through the observation module. An example of an observation module is described in this chapter.

Chapter 7 This is the description of the actual tool for debugging for logical bugs. Through the analysis module (which we call the tool), a trace can be examined for logical bugs by the use of visualizations.

Chapter 2

Observation of distributed applications

With the goal of finding a solution of the global timing problem, we introduce the use of logical time and partial orders as an approach for observing distributed applications.

First Lamport's classical definition of partial ordering of events is defined. Then a time stamping algorithm is given. This algorithm makes use of logical clocks for partially ordering events, and was also proposed by Lamport.

Based on Lamport's theory we describe an extension of the concept of logical time proposed by Fidge which gives an implementation of the partial ordering more useful for debugging distributed applications.

2.1 Ordering events partially

In order to solve the problems of observing distributed applications, Lamport defined in [Lam78] a partial ordering of events:

> The relation $\rightarrow$ on the set of events of a system is the smallest relation satisfying the following three conditions:
>
> 1. *Sequential behavior*. If a and b are events in the same task, and a occurs before b, then $a \rightarrow b$.
> 2. *Asynchronous message-passing*. If a represents the sending of a message and b represents the reception of the same message, then $a \rightarrow b$.
> 3. *Transitivity*. If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$. Two distinct events a and b area said to be *concurrent* if $a \not\rightarrow b$ and $b \not\rightarrow a$.

The $\rightarrow$ relation is named *happened before*. The definition is remarkable in several ways. Concurrent events are not arbitrarily ordered and the definition only relates two events a and b if one event could affect the other.

If $a \rightarrow b$ then event a can effect the event b. Therefore the relation is called the *causal relation*. We say that the event a can casually effect the event b. In the sections to come we will use this terminology when arguing about relations between events from a generated trace.

We will use this definition to form a basis for problems concerning observation of distributed applications.

2.2 Lamport time

In his article, Lamport defined a distributed algorithm for time-stamping all events in such a way, that the time-stamps follows the $\rightarrow$ relation. The algorithm can be seen as an attempt to solve the observation problem related to lack of global timing.

2.2.1 Algorithm

The idea in Lamport's algorithm is that each task maintains a logical clock, that is used for time-stamping events. The algorithm is quite simple and can be summarized as follows:

1. *Logical clock.* Every task p_i is associated with the counter C_i, which represents the logical clock for this task. This is also called the Lamport clock.
2. *Initialization.* The logical clock is initially 0.
3. *Ticking.* When the task p_i creates an event e, C_i is incremented at least once.
4. *Monotonically increasing counters.* No counter is ever decremented.
5. *Sending.* When the task p_i sends a message, then C_i is first incremented according to rule 3 and C_i is then piggy-backed with the message.
6. *Receiving.* When the task p_j receives a message from the task p_i, C_j is set to $\max(C_j, v + 1$, where v is the value piggy-backed with the message.
7. *Time-stamping.* All events are time stamped with C_i after the appliance of rules on the event. The time-stamp of an event e is written as $C(e)$; we also write $e(t)$ to indicate that event e is time-stamped with the time t.

An example of the use of logical clocks for time-stamping is seen on figure 2.1. Based on the information from the logical time-stamp of each event, Lamport forms the following condition, also called the clock condition:

$$\forall\ e_i, e_j : e_i \rightarrow e_j \Rightarrow C(e_i) < C(e_j)$$

If there exist the *happened before* relation between any two events e_i and e_j then we are able to reason about the values of the corresponding time-stamps.

2.2.2 Evaluation

Through the use of logical time-stamps we are able to reason about the relationship between events according to the definition of partial order. However, Lamport's implementation has significant limitations. The clock condition goes only one way; if there exists a partial order, we can reason about the corresponding event time-stamps. Not the other way. This means that the algorithm is not usefull for observing distrubuted applications, because we can not examine arbitrary events and reason about their causal relation.

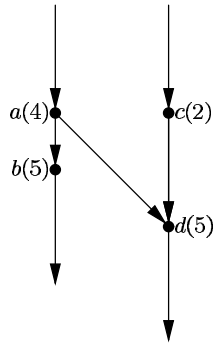


Figure 2.1: Events time-stamped with Lamport's logical clocks, the time-stamps now reflect the causal relations between events.

Furthermore, the definition does not deal with concepts such as task creation, task termination or synchronous communication. As a consequence the area of use narrowed. In the next section we will examine another approach for partially ordering events.

2.3 Vector time

In the previous section it was found that Lamport time has certain limitations. The limitations lie in the lack of ability to reason about events causal relation. In this section we introduce vector time which gives this ability.

Vector time gives the ability to reason about events causal relation. This is done by increasing the amount of information available in the time-stamps. The extra information represents the knowledge that a task has about the clock in the other tasks of the application. The extra information makes it possible to see whether any two events have a causal relationship or not. Furthermore, vector time is defined on a broader definition of partial order than Lamport's algorithm, which widens the area of use. The definition of vector time that is described in this section was developed by Fidge in [Fid91].

2.3.1 Partial order

Fidge gives a definition of a partial order that handles synchronous communication and task creation and termination. We extend the definition described in section 2.1 to suit this. This means that the definition proposed by Lamport is a subset of this revised definition:

4. *Task creation.* If event e and task q occur in task p, event f occur in q, and q begin after e, then $e \rightarrow f$.

5. *Task termination.* If event e and task q occur in task p, event f occurs in q, and e occurs after q terminates, then $f \rightarrow e$.

6. *Synchronous message-passing.* If event e is a synchronous input (output) and event f is the corresponding output (input), and there is an event g such that $e \rightarrow g, then f \rightarrow g$. If there is an event h such that $h \rightarrow e$, then $h \rightarrow f$.

Figure 2.2 illustrates the definitions for task creation and termination.

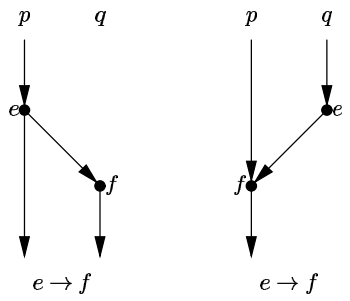


Figure 2.2: The definitions for task creation and termination.

2.3.2 Algorithm

The idea of vector time is that each task in the computation maintains a vector. This vector correspond to the logical clock in Lamport's algorithm except that is holds more information. The following first describes the rules for maintaining the vector clock. Then we define the $<$ operator and prove how it can be used to compute the $\rightarrow$ relation. After this an example-of-use is provided.

In order to describe the rules for maintaining vector-time, it is necessary to define the function max. This function is used to calculate a vector consisting of the maximum values of a given set of vectors.

$$\max\{C_1, \ldots, C_n\} = \begin{bmatrix} \max(C_{1,1}, \ldots, C_{n,1}) \\ \vdots \\ \max(C_{1,m}, \ldots, C_{n,m}) \end{bmatrix}$$

where C_n is nth vector and $C_{n,m}$ is the mth entry in the nth vector.

The following rules describe the algorithm for maintaining vector time:

1. *Vector clock*. Every task p_i is associated with a vector C_i that has a dimension equal to the total number of tasks that are going to exist during the computation. The ith element corresponds to the Lamport clock for p_i. The entry $C_{i,j}$ where $j \neq i$ denotes p_i's knowledge of p_j's time.

2. *Initialization*. The vector initially associated to a task is the null vector.

3. *Ticking*. Whenever a task p_i performs an event, $C_{i,i}$ in incremented at least once.

4. *Monotonically increasing counters*. In a computation with the set of tasks $p_i, \ldots, p_l$, no entry $C_{i,n}$ where $n \in i, \ldots, l$ is ever decremented.

5. *Task creation*. Whenever a task p_i creates a set of tasks $p_j, \ldots, p_l$, they each inherit the vector of the parent:
$$\forall x \in \{j, \ldots, l\} : C_x := C_i$$

6. *Termination*. Whenever a set of tasks $p_j, \ldots, p_l$ terminates, the parent p_i sets C_i to:
$$\max(C_i, C_j, \ldots, C_l)$$

7. *Synchronous events*. During a synchronization, all tasks $p_j, \ldots, p_l$ involved exchange clocks with each other, that is they synchronize their watches:
$$\forall x \in \{j, \ldots, l\} : C_x := \max\{C_j, \ldots, C_l\}$$

8. *Sending*. Whenever a task p_i sends a message, it is piggy-backed with C_i. A task only applies this rule after any increments required by rule 3.

9. *Receiving*. Upon receiving a message from p_j with the piggy-backed vector C_j, the receiving task p_i sets C_i to:
$$C_i = \max(C_i, C_j)$$

10. *Time-stamping*. All events in P_i are time-stamped with the clock C_i after the appliance of rules on the event. The time stamp for the event e is defined to $C(e)$ and the nth element of the time stamp is denoted $C_n(e)$.

It should be noted that the algorithm for vector time is strong enough to allow communication like broadcasting, multicasting and barriers. Also it is not a requirement that messages are delivered i FIFO order, which is often the case in distributed algorithms.

In order to compute the $\rightarrow$ relation, we define the $<$ operator, and prove that it is equivalent with the $\rightarrow$ relation.

Given the event e from the task p_i and an event f from the task p_j. The time stamps $C(e)$ and $C(f)$ are then compared with the comparison operator $<$. Fidge defines the two operators $\leq$ and $<$ to:

$$C(e) \leq C(f) \quad \Leftrightarrow \quad C_i(e) \leq C_i(f) \tag{2.1}$$
$$C(e) < C(f) \quad \Leftrightarrow \quad C(e) \leq C(f) \ \wedge \ C_j(e) < C_j(f) \tag{2.2}$$

Assume that $C_i(e) \leq C_i(f)$ holds. Then e must have happened before or at the same time as f:

> p_j's knowledge of p_i's clock at the time f occurs is more resent than p_i's clock at the time e occurs.

This could make us believe that $e \rightarrow f$, iff $C(e) \leq C(f)$. But then $e \rightarrow e$ because $C(e) \leq C(e)$, which is nonsense. Because of reflexivity we can assume that $e \rightarrow e$; then because of transitivity we can also assume that $e \rightarrow f \rightarrow e$ where f is an other event with the same time-stamp as e, that is e and f are synchronous. This means that the following holds:

$$C_i(e) \leq C_i(f) \wedge C_j(f) \leq C_j(e) \tag{2.3}$$

The first expression covers the fact, that $e \rightarrow f$, and the second expression covers that $f \rightarrow e$. We ensure irreflexivity, that $f \not\rightarrow e$, by negating the second expression. This gives us the definition of the $<$ operator shown in 2.2. The operator has the following property:

$$e \rightarrow f \Leftrightarrow C(e) < C(f)$$

This means, that with the use of the $<$ operator, it is possible to effectively calculate the $\rightarrow$ relation. Proving that the $<$ operator actually holds requires proof by structural induction, which we find to be beyond the scope of this report. This definition of the $\rightarrow$ relation is quite close to the clock condition defined in section 2.2. The difference is that with Lamport's clock condition it is only possible to reason about time-stamps if there exists a partial order. Not the other way.

Figure 2.3 shows the space-time diagram for an example computation. Checking the causal relationship between events is quite easy when the $<$ operator is used. In the following we will compare three events from the example.

- test if $e \rightarrow e$:
$$(1 \leq 1) \wedge (1 \not< 1) \Rightarrow \neg(C_{T_0}(e) < C_{T_0}(e))$$

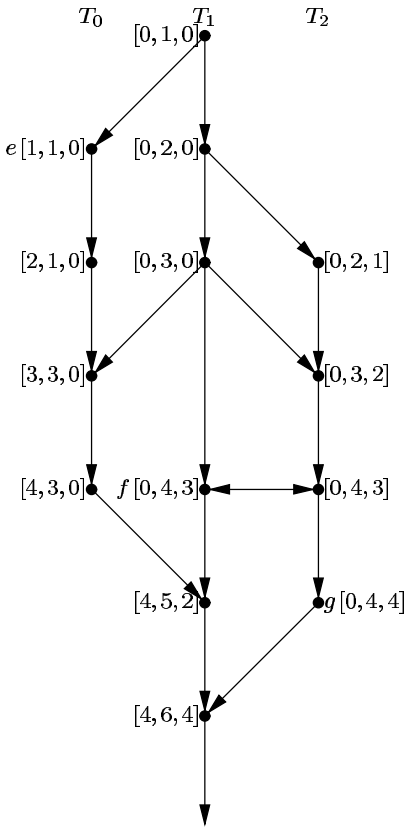


Figure 2.3: A space-time diagram for a distributed computation involving three tasks. Each node in the graph represents an event, and each event is decorated with a vector time.

- test if $e \rightarrow g$:
$$(1 \not\leq 0) \wedge (0 < 4) \Rightarrow \neg(C_{T_0}(e) < C_{T_2}(g))$$
- test if $f \rightarrow g$:
$$(4 \leq 4) \wedge (3 < 4) \Rightarrow (C_{T_1}(f) < C_{T_2}(f))$$

2.3.3 Evaluation

When comparing events time-stamped with vector time, it is possible to check whether the events are causally related or not. Lamport's logical time only has the information that there can exist a causal relation between the events. This means that vector time gives a more detailed view of the application being debugged.

In a distributed debugging context, vector time can be used as a tool for observation of the causal relationship between events. This is important in understanding the behavior of the application. If a programmer knows that two events have a causal relation, then one of them can influence the other and thereby be the reason for a logical bug.

The problem with vector time is the lack of scalability, since the memory used for holding time-stamps during execution is at least quadratic in the number of tasks, and this memory usage will grow fast with

the number of events.

Observing an application with vector time, will impose quite a large amount of perturbation on the application. The unmodified algorithm require that all messages are piggy backed with a vector, which will give a communication overhead. Any optimizations of the memory usage will be on the price of a processing overhead, where vectors are compared and dynamically allocated. Also if the number of tasks that are instantiated during application execution is not known, then the vector size is indeterminable, and therefore the size can change dynamically.

2.4 Conclusion

In this chapter we have argued for the use of logical time as an approach for finding a solution to the observation problem associated with lack of global time. We have described two methods for achieving a partial ordering of events.

It was found that vector time gives an appropriate level of detail, suiting the need for locating logical bugs, thereby making it more useful in a debugging purpose. The problem was that the level of detail was on the price of perturbation.

Chapter 3

Debugging with causal graphs

In the previous chapter we argued for the use of a partial ordering of events, in order to solve the global time problem in debugging distributed applications. Despite the fact that we solve this problem, the complexity problem has not been addressed yet. As mentioned in chapter 1, the goal for this project is to handle the complexity problem by the use of visualization techniques. In this chapter we will describe how the communication patterns of an application can be visualized in a graph called the *causal graph*.

Fidge has proposed an algorithm for stamping events with vector-time and an comparison operator, that efficiently computes the causal relation between two arbitrary events by using the time-stamps.

As described in 2.3.3 this algorithm is quite complicated and has the serious problem that it lacks scalability. We will take another approach in this chapter. An algorithm that decorates nodes (that is, events) in the causal graph with vector-time will be developed. We will describe how the causal graph can be used for debugging and how the decoration will enable us to define the same comparison operator as Fidge, without the need for time-stamping with vector-time during execution.

3.1 Definition of the causal graph

The causal relation $\rightarrow$ defined in 2.1 and later refined in section 2.3.1 makes it possible to test the causal relation between two events. In order to visualize the communication pattern of an distributed application, we need a more fine-grained frame-work.

Given an event, it is possible to compute all events happening before, after or concurrently with the event. What is just as interesting, is to compute immediate predecessors and successors of the event since this represent the actual communication patterns, i.e.:

- Given an event from task p, calculate the previous or next event occurring in task p.
- Given a send event, calculate the corresponding receive event.
- Given a receive event, calculate the corresponding send event.

When using vectortime these computations cannot be done efficiently. We therefore organize all events in a causal graph. A causal graph is a directed acyclic graph (DAG) where a node represents an event, and a directed edge from e to f represents the fact, that f occurs immediately after e. That is, the edge represents, that e *caused* f to occur. In a MPS each asynchronous communication, task creation or task

termination corresponds to an edge in the causal graph. There will also be an edge from e to f, if both events occur in the same task immediately after each other.

Because of the way we defined $\rightarrow$, there is a close correspondence between the causal graph and the causal relation. Actually they are equivalent:

> $e \rightarrow f$, iff there is a path from e to f in the causal graph.

The main difference is that the transitive rule (rule 3 in section 2.1) is used in computing the causal relation—a corresponding rule is *not* used in computing the graph. In principle it is not necessary to have both $\rightarrow$ and the causal graph computed, since they contain the same information. In practice, they both have the property of being efficient when implemented:

- When using vector-time as proposed by Fidge the causal relation gives efficient comparability of arbitrary events for causality.
- The causal graph gives efficient access to predecessors and successors to an event.

Therefore a combination of the two is most feasible: having events partially ordered and structured in a causal graph.

In the next sections, the use of causal graphs in debugging will be argued. Algorithms for building the graph and stamping events in the causal graph with vector-time will be described.

3.2 Using causal graphs in debugging

Since we have focused our attention on locating logical bugs, it is necessary for the visualization to represent the logic of the application—in MPS's this means the patterns in which tasks affect each other. The causal graph directly corresponds to these patterns—therefore it is potentially useful for debugging.

If a bug occurs in a task, the bug can spread to other tasks by following causal relations—or edges in the causal graph. In the causal graph, it is possible to follow paths emanating from an event under the suspicion of being buggy. Likewise it is possible to go the other way—locating events that could have caused a buggy event. This simply requires following paths ending in the event backwards.

Even though the causal graph is good for following casual relationships, it still does not handle the complexity problem. The causal graph does not reduce the number of events in the trace—it merely introduces a structure which makes it easy to navigate stepwise between events. We are very close, though. As argued above, the user would follow paths emanating or ending in a buggy event in order to locate the cause of or damages caused by the bug. This also means, that the user probably has no interest in following paths not having this property—his search would actually be simplified significantly if they were removed from the graph. We therefore believe that the following properties of a visualization would reduce the complexity of debugging the application:

1. The visualization consists at all times of a subgraph of the causal graph.
2. Given an event, it is possible to expand the visualized subgraph with all events causally related with the event.
3. Given an event, it is possible to reduce the visualized subgraph by removing events causally related with the event.

This way the user can remove unwanted information if it clutters the visualization, and he can add it again later, when it becomes necessary. The subgraph initially visualized must contain enough events to reach all other events in the causal graph—this is all events having the following properties:

1. The event is the first event from a task and the event has no predecessors.
2. The event is the last event from a task and the event has no successors.

By using the characteristics of the causal graph through the above functions for reducing and expanding visualized subgraphs the debugging activity can be made less complex. It could be argued that other functions using the casual graph could be usefull. For now we will limit the functions for visualization to above.

3.3 Building the causal graph

In the following we will describe the algorithms we have designed for building the causal graph and decorating events in the graph with vector time-stamps.

3.3.1 Building the graph

Building the graph can be done using almost the same techniques as the algorithm proposed by Fidge for building the causal relation. For each of Fidge's rules, there is a corresponding rule for connecting two nodes in the causal graph with an edge. Despite this, we have developed an even simpler and more efficient approach, using a modified version of Lamport's time-stamping algorithm. This algorithm will establish the same causal relations between events as using Fidges algorithm and enable similar possibility for reasoning about relations between events.

In the following we will describe this algorithm, and we will assume asynchronous communication—the case of synchronous communication will be handled in section 3.3.2.

In order to make an edge from an event e to an event f, one of the following properties must hold:

1. f occurs immediately after e in the same task, that is, f is the event in the task with the smallest time-stamp greater than $C(e)$.
2. e represents the sending of a message, and f represents the reception of the message.
3. e represents the creation of a new task, and f represents the first event in the new task.
4. e represents the termination of a task, and f represents the matching event in the parent task.

The four cases are handled differently. In the following we will call events like e for source events and events like f for destination events.

Assume that we stamp each event with Lamport time (as described in section 2.2) and the identity of the task from which the event origins. It is then a simple task to make all edges in the causal graph that come from property 1. Moreover if we assume that events from one task are in FIFO order, it reduces the necessary bookkeeping to a minimum.

In order to make edges on behalf of properties 2–4, we extend Lamport's time stamping algorithm. On each destination event participating in a communication, we also stamp the Lamport time and task identity from the source event. This means that it is necessary to piggyback all messages with both

time and task identity. Using these extra stamps, it is straightforward to relate the source event with the destination event.

The rules are described as follows:

1. *Logical clock and task identity*. Every task p_i is associated with the counter C_i, which represents the logical clock for this task.
2. *Initialization*. The logical clock is initially 0.
3. *Ticking*. When the task p_i creates an event e, C_i is incremented at least once. An edge is generated from the event with the value before ticking to the event receiving the new logical time.
4. *Monotonically increasing counters*. No counter is ever decremented.
5. *Sending*. When the task p_i sends a message, then C_i is first incremented according to rule 3 and (p_i, C_i) is then piggy-backed with the message.
6. *Receiving*. When the task p_j receives a message from the task p_i, C_j is set to $\max(C_j, v+1)$, where v is the piggy-backed logical time sent with the message. The event with timestamp (p_i, v) is matched with the event with timestamp $((p_j, C_j), (p_i, v))$ and an edge is generated these events.
7. *Time-stamping*. All events e from task p_i are timestamped with task identity and logical time $((p_i), (C_i))$. When messages are sent, the piggy-backed information is supplied with the timestamp. All events are time stamped after all other rules have been applied.

In figure 3.1 is shown an example of how a causal graph is created. By using the extended stamping algorithm, it is easy to match source events with destination events. It is noted here that it is necessary to categorize all events in the MPS used, since they are stamped differently. The algorithm is thus very close to the MPS in question.

When using this algorithm, the underlying MPS is slightly more perturbated than when only Lamport's time stamping is used. All messages are piggy-backed with double the amount of data. Since the cost of maintaining Lamport clocks is minimal, we do not regard the increased perturbation as significant. In comparison to Fidge's algorithm, the algorithm is less perturbative, since no vector-times are piggy-backed. The memory consumption is smaller too, since tasks only uses a Lamport clock instead of a vector-clock.

3.3.2 Handling synchronous communication

Until now we have done nothing to handle synchronous communication, since the causal relation we have used does not distinguish between concurrency and synchrony. Since synchronous events happen at the same time, it is important that preceding and succeeding events are *similarly causally related* to the synchronous events.

In order to ensure that synchronous events happen at the same time, we extend Lamport's algorithm to ensure this. This adds a new rule to the algorithm in from previous section (Se figure 3.2):

8. *Synchronization* For events e and f in tasks p_i and p_j make edges from all predecessors $(e_{p,1}..e_{p,n})$ of e to f and from all predecessors $(f_{p,1}..f_{p,n})$ of f to e. Similarly for successors: make edges from e to all successors $(f_{s,1}..f_{s,n})$ and make edges from f to $(e_{s,1}..e_{s,n})$.

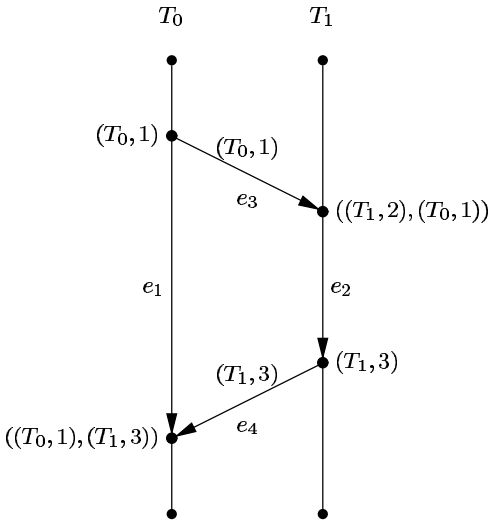


Figure 3.1: This figure illustrates a client-server example. Here the edges e_1 and e_2 corresponds to edges made because of property 1, while edges e_3 and e_4 corresponds to edges made because of property 2. The vectors shown at edges e_3 and e_4 are the piggy-backed time and task identity. At each event timestamps are shown as two dimensional vector indicating task identity and logical time. Destination events are also stamped with the the vector from the piggy-backed information.

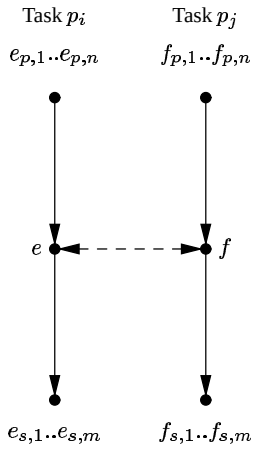


Figure 3.2: The rule for extending the causal graph with synchronizing events adds extra edges from predecessors and successor to the participating events..

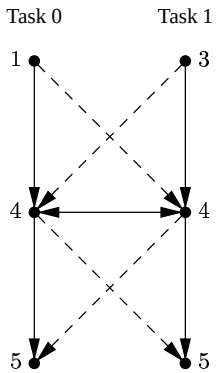


Figure 3.3: This figure illustrates how the time stamps are assigned at synchronous events to the maximum time of the participating events. In order to ensure the causal relation as defined by Fidge, the dashed edges are added—this ensures that predecessors and success to synchronous events are similarly causally related.

An example of this rule being applied is seen in figure 3.3. In the figure, it is seen, how edges are added to the graph in order to ensure the causality. This way the predecessors for the event at time 4 in task 0 are also predecessors for the event at time 4 in task 1. Thus the causal graph holds the information that synchronous events happen at the same time. The same holds for the successors to the synchronous events.

When considering visualization of synchronous communication it is not necessary to actually visualize the additional edges needed in the causal graph. These are only inserted to ensure the causal relations.

3.3.3 Adding vector time to the causal graph

Checking the causal relationship between any two events in the graph can be used for gaining insight in a larger graph. Such a check will in the worst case cost a complete search of the graph. By decorating the graph with vector time it is possible to check the causal relation between any two events in constant time. Therefore we chose to give an algorithm for decorating the causal graph with vector time well aware of the scalability problem with vector-time.

The algorithm is a complete run through the graph based on Fidge's vector-time algorithm from section 2.3.2. The algorithm iterates continuously between tasks until the hole graph is decorated. Given a causal graph, where each task has a cursor that points to an event for that task then the graph is decorated by applying the following rules:

1. *Initialization*. If there is no preceding event then assign a vector time to the event according to the vector time rules, and move the cursor to the next event in the task.
2. *Time-stamping*. If all predecessors have a time-stamp, then time-stamp the current event according to the vector time rules, and move the cursor to the next in the task.
3. *Wait and release*. If any of the preceding events do not have a vector time, then decorate the next task. Do not assign any vector time to the current event, and do not move the cursor.
4. *Release*. If the last event for the task has been assigned a vector time then decorate the next task.
5. *Stop*. If the last event in all the tasks has been assigned a vector-time then stop.

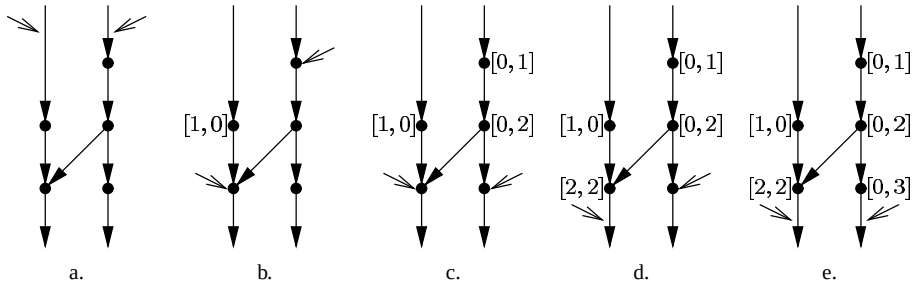


Figure 3.4: The figure is a scenario of how the algorithm decorates the causal graph with vector time. The unfilled arrows represent the cursors.

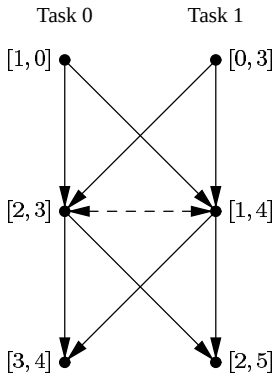


Figure 3.5: This example illustrates how the extra edges inserted for handling synchronous communication ensures a consistent decoration of a causal with vector-time.

The algorithm will run in $O(n)$, where n is the number of events in the graph. Figure 3.4 shows a scenario of how the algorithm works.

The inclusion of synchronous events in the causal does not limit the decoration of vector-time. We prove this by giving an example. This is shown on figure 3.5. The extra edges ensures that the causal relations are preserved. Events before the synchronization are causal related to events after synchronization. This can be tested both by following edges and by testing vector-time using rules proposed by Fidge. The extra edges have also preserved parallelism of the synchronizing events. This again can be tested using the rules for checking causal relations.

3.4 Conclusion

In this chapter we have argued for the use of causal graphs in debugging. The causal graph lends itself naturally to visualization, because it shows the communication patterns in an application. It was shown the the causal graph is equivalent to the causal relation. This means that it is not necessary to compute both of these, in order to use them both. We presented an algorithm that, given a trace from a distributed application, created the causal graph using rules very close to the rules used in Fidge's algorithm for stamping events with vector-time. This algorithm was extended to handle synchronous communication. Since vector time-stamps enables efficient comparisons of events for causal relation using Fidge's comparison operator, we presented an algorithm that, using the causal graph, decorated all events with vector time-stamps.

Compared to the theory developed by Fidge we have achieved the same possibilities for precise reasoning about application behavior. However we achieve by the use of algorithms which are much more efficient and less disturbing for the application being observed by. This is done by building causal relations only using Lamport time and task identity and adding vectortime to events postmortem. In addition the causal graph enables functions for visualization which can reduce complexity.

Chapter 4

Architecture

The theories developed in the previous chapters form the basis for the development of a debugging tool. We name this tool *traceinvader*.The architecture of the *traceinvader* system is described in this chapter. The goal of the architecture is to ensure a flexible solution that allow support of the overall requirements described in section 1.4.

Figure 4.1 is an illustration of the architecture, which consists of three modules: the trace storage module, the observation module and the analysis module. The modules communicate through well defined interfaces securing the flexibility of the architecture.

The following describes the overall aspects of each module, followed by a description of the data-flow in the system.

4.1 Trace storage module

The idea is that this module should make out the backbone of the system, by providing as much general functionality for generating traces as possible. We wish to support all kinds of MPS'es. The functionality of this module can be separated into functionality for building, access, and storing a trace.

Building a trace is supported by an advanced interface, which provides general functionality for building traces that fulfill certain requirements. These requirements are necessary in order to secure a consistent trace. A main goal is to hide the specific functionality for storing traces.

Access to the trace can be used by both the observing and analysis module. The goal is to provide a general interface that works independently of the underlying storage model.

Since traces tend grow very large, a sophisticated storage model is needed. The storage model is to

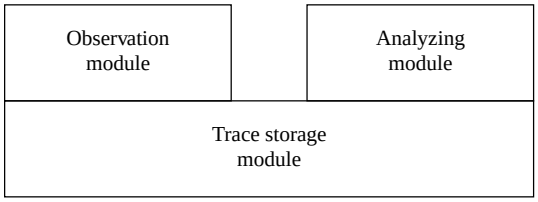


Figure 4.1: The *traceinvader* system architecture.

provide as efficient access to the trace as possible through the provided interfaces.

4.2 Observation module

The observation module is dedicated to the MPS in which the application being debugged is implemented. It uses the interface provided by the trace storage module and information specific for the MPS for building the actual trace.

The observation module is to help fulfill the first part of the hypothesis described in section 1.4 by ensuring that the observation behavior of an application is reliable.

Generality of the system is secured because the observation module is the only module that is changed when porting a MPS to the system. In addition the interface supplied from the trace storage module keeps the work of making an observation module to a minimum.

A programmer of a distributed application will not need to know any details of the observation module. Only knowledge of the enabling and disabling of the tracing facility is needed.

4.3 Analysis module

The analysis module supplies the debugger with a user-interface for post-mortem analysis. It is through the analysis module that the programmer gains an overview of the application behavior and is able to locate bugs.

The main goal for the analysis module is to fulfill the second part of the hypothesis for reducing complexity described in section 1.4 . This is done by providing an intuitive user-interface for gaining an overview of the application by means of visualizations. Several different visualizations are provided. The central visualization shows the causal graph for handling the complexity problem. Alternative visualizations work as support for the graph. These are an animation, support for static debugging, and the ability to relate a trace to the source code.

4.4 Data flow

The data-flow of the architecture shown in figure 4.2 describes how the system is integrated into a debugging context.

A scenario that follow the data-flow of the architecture is when a programmer is developing a distributed application. The observation module observes the behavior of the application during execution. The observed behavior is then used by the observation module for building and storing the trace using the trace storage module. The trace storage module provides the analysis module with a trace. The programmer will then analyze the trace and maybe locate a bug. With the understanding of occurred bugs the programmer works out a scheme for removing bugs which is implemented in the distributed application.

4.5 Conclusion

In this chapter we have argued for and explained the overall architecture for the *traceinvader* system. We believe that this solution will provide a flexible basis for the debugger.

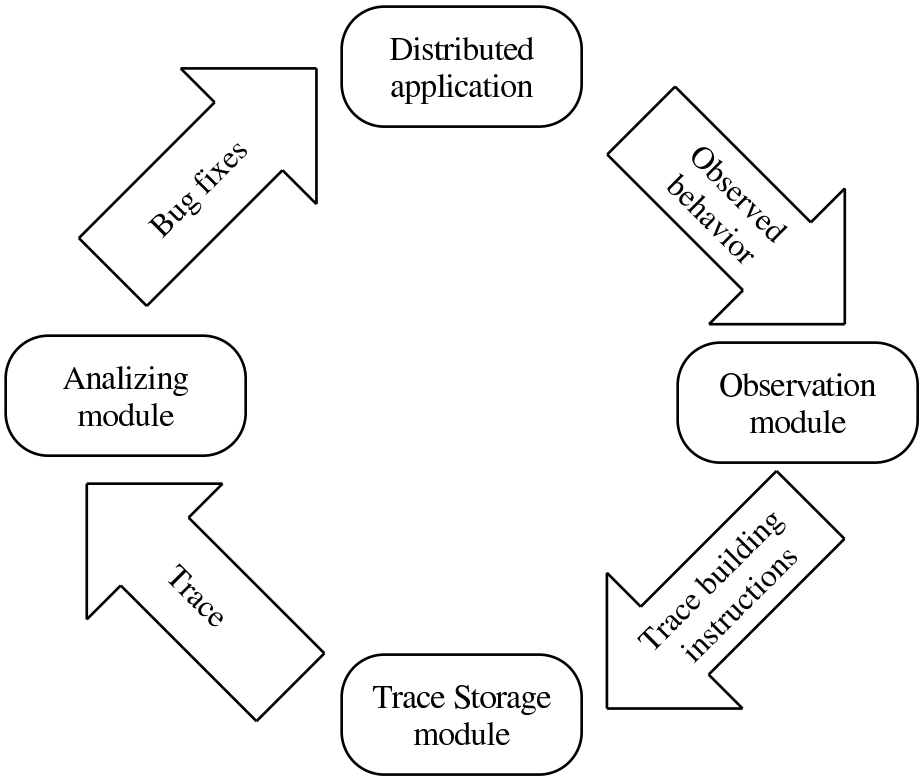


Figure 4.2: The data-flow of the architecture.

In the following chapters each of the different modules will be described in detail.

Chapter 5

Trace storage module

As described in chapter 1 we require the debugger to be based on postmortem trace analysis. The trace storage module is to provide functions for generating, storing and accessing a trace, which can be used in the analysis module.

The use of causal graphs for debugging and the need for supporting traces generated from different MPS's forms the basis for the module developed in this chapter. In the following sections we state requirements for the design and give a more detailed description of how this module is to be used together with the analysis and observation modules. After this, a detailed design of the trace storage module is described.

5.1 Design goals

In chapter 1, a trace was defined as a sequence of events recorded from a distributed application. Having introduced the use of causal graphs for debugging in chapter 3, we extend the definition of a trace. A trace is a set of events structured in two ways:

1. As tasks, i.e. sequences of events.
2. As a causal graph.

The trace storage module must support this definition and provide functions for building, storing and accessing such a trace.

The use of causal graphs for debugging supports the requirement for generality, since a casual graph is not limited to a specific message passing paradigm. A partial ordering of events can be enforced on any distributed application using message passing.

To generate a trace from any kind of MPS, the trace storage module needs to provide generally usable functions for storing events. An event collected from a distributed application represents some action of interest, so functionality is needed for handling and storing the information which identifies different types of events.

Given the above needs we state the design goals as follows:

Event modelling Depending on the specific MPS being used, different information need to be attached to events. The trace storage module is to support these varying needs by enabling modelling of

the characteristics of different MPS's.

Trace building During the observation of a distributed application, events are generated. These events are to be used in the construction of a trace. The trace storage module is to support the generation of events and the structuring of these as defined above.

Generality An important property of the trace storage system, is that it provides standard interfaces for handling traces, so they can be generated from any observation module and used in the analysis module.

5.2 Modelling traces

Modelling of events is ensured through the use of the object-oriented programming paradigm. Collected events are objects, which are instances of classes describing events. In addition to model events and their properties, the object-oriented paradigm gives us the strength of using inheritance, which simplifies the process of modelling significantly. By providing standard classes describing basic properties common to all MPS's, these can be used whenever possible, and specializations can be made otherwise.

Based on these considerations, the trace storage module must provide functionality for creating and accessing classes describing different event types and their properties. The module must also provide a basic hierarchy of classes describing standard events common to all MPS's. Creation of events is done through functionality for creating objects, i.e. instances of classes.

The trace storage modules provides the functionality for creating the structure which makes out the trace. Thus, it must provide functionality for creating tasks and causal graphs. To support the algorithms for building a causal graph as described in chapter 3, each event must be supplied with a logical timestamp and the identity of the task in which the event occurred. These are also used when structuring the events as tasks.

The storage of traces is provided by the trace storage module through functionality for saving objects and classes. In addition to this, the storage of the causal graph and task structures must be supported by the module. Functionality for later retrieval must likewise be provided.

5.3 Usage

With the above description of the mechanisms provided for trace generation we now give an more detailed description of the context in which this module is to be used.

After having generated the necessary hierarchy of tasks and event types for a specific message passing paradigm, the next phase represents the insertion of statements (see figure 5.1) in a specific distributed application. This instrumentation ensures events to be generated during runtime. As described the observation module is responsible for this instrumentation.

In principle the insertion of event generating statements could be automaticly done through a useful preprocessing script. Or the appropriate statements could be inserted by hand directly in the sourcecode.

The last phase represents the observation of the distributed application and processing of generated into events as illustrated on figure 5.2. During runtime the application emits events reflecting the behavior. The generic trace system is not intended to provide the actual collection of events from the application. This again is the responsibility of the observer module.

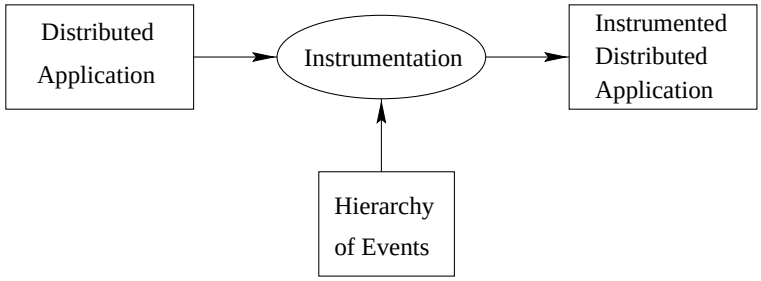
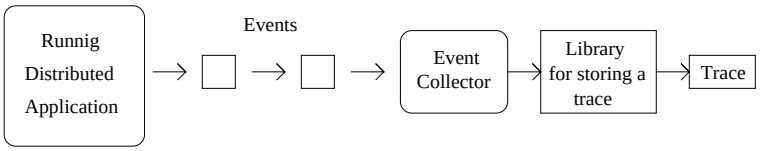


Figure 5.1: Instrumenting the source code of a distributed application with event generating statements.



gure 5.2: Collecting events from a distributed application and using the generic trace system to store e trace.

ıe trace-system is only to supply methods for building the causal graph. This is done by providing a ecial purpose library for making instances of event types, insertion of these events in a causal graph d for storing of the trace. This construction of graphs is enabled by using a standard interface. The ›servation module is to ensure that the graph being built is consistent with correct causal relations.

y using the object system, the trace storage module ensures a consistent storing of a generated trace. ıce the causal graph has been built using a supplied interface, the generated trace is stored in a way can reproduced for postmortem visualization. When the analysing module needs to visualize a gen- ate the storage module is responsible for reproducing the causal graph and supplying the information tached to events.

.4 Design

e separate the design description of the trace storage module in two parts. First we describe the object ucture which is to support modelling of different types of events and task and secondly we describe ›w persistence of the traces are to be handled. The purpose of following sections is not to describe the ıce storage module at implementation level but give an understanding of how we have chosen to meet e requirements through object-oriented models and storage capabilities.

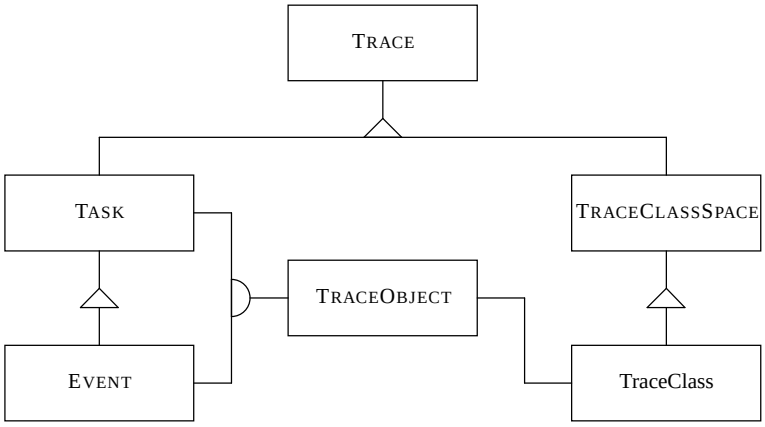


Figure 5.3: The structure of object system, that provides functionality for dynamically defining classes that are specializations of the basic task or event.

A class describes a type through a set of named properties or attributes. Properties can be inherited from a superclass, hereby making a specialized class of the superclass. A class can only inherit directly from one other class. The properties must have different names.

An object is an instance of a class. The relation between the class and object is that the object holds the values to the properties of the class. This way there can be many instances of a class.

Obviously the best way to provide modelling of tasks and events would be through an environment that provide dynamic class creation. This is not directly possible since we have chosen to base the modules on more static platforms, see sections 3.3.1 and 5.1. The problem has been solved by providing the functionality for dynamic class definition through a number of classes, that describes the object system. The structure of this is show on figure 5.3.

The functionality for dynamic class definition can be provided simply because of the simplification which only allows inheritance of attributes.

A TRACE contain a TRACECLASSSPACE. The TRACECLASSSPACE is the container which holds the different class definitions. TRACECLASS holds the definition of a class, by having a list of properties, and a reference to the superclass. The TRACECLASSSPACE is used for access and creation of classes.

TRACEOBJECT symbolizes a class instance and provides a reference to the TRACECLASS, hereby giving the object a type and making it possible to connect property names and values.

TASK and EVENT are specializations of TRACEOBJECT. These two classes are the two general classes that provide functionality for structuring the causal graph and the trace storage module provides functionality to make specializations of these.

Through specializations of TASK and EVENT the task and event types can be tailer-made to fit observation of any kid of MPS. Functionality for attaching extra information to events in the causal graph is provided by adding attributes to event types.

The trace storage module it to provide a default hierarchy for tasks and event types. This hierarchy defines TASK and EVENT with a minimum set of properties. Figure 5.4 shows the two default hierarchy's. A task always has a task identifier called *tid* and an alias, *alias*. The event hierarchy is extended so that a PRINTEVENT is also provided. This event type is to support a form of static debugging. An event has four basic properties, these are *time* which is the logical time for the event, *tid* which defines the task relation for this event. The properties *file* and *line* makes it possible to locate the source code location

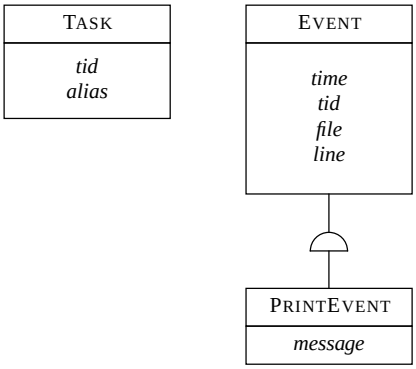


Figure 5.4: The figure shows the default class hierarchy for the trace system..

for the event.

This hierarchy provides support for building the causal graph from the information attached to events. This is done using *time* and *tid* and as described by the algorithms in chapter 3.

5.4.2 Storage

The trace storage module is to provide a bridge between the observation module and the analysis module. When observing the behavior of distributed applications the traces generated can grow very large because of the state explosion. This calls for a sophisticated storage model.

We have chosen to make the persistence model as simple as possible, and still provide the basic functionality. This is done by storing the trace into three separate sequential access files. The general problem of storing a trace with this approach is that any forward dependencies will complicate the retrieval with sophisticated administration.

The first file contains the class hierarchy that was build dynamically during the event collection. The dependency problem is handled by placing the general classes first. The second file is a file of source code filenames. This file does not depend on other files. The third file is the actual trace.

The trace file is organized into three parts. First part consist of all the different TASK objects. The second part consist of all the EVENT objects. The events are placed so that all the events from each task are placed together placed in their causal order. The last problem is storing all the edges between the tasks in the causal graph. These need to be sorted so that no search of the graph is needed when retrieving the graph.

Storing the trace is quite complicated, because a total order is enforced on the partial order of events. Once a trace has been stored and it is to be analysed postmortem in the analysis module, this module provides functions for establishing the partial order between events by reproducing the causal graph.

5.5 Conclusion

This chapter has described the trace storage module responsible for generating a trace and reproducing the same trace for visualization in the analysis module. This module ensures the generality need for handling observation different message passing systems. By building causal graphs and storing traces through a standard interface the behavior of a distributed application can be used for visualization in

the analysis module. The trace storage module provides the necessary access to traces by providing functions for reproducing the causal relation between events and the information attached to events.

Chapter 6

Observation module

In this chapter we describe a prototype implementation of the observation module. The goal is to test our theories for building traces and to gain insight into the problems of developing an observation module. The procedures for using the module was described in chapter 5.

The prototype described provides instrumentation for a specific MPS called the Parallel Virtual Machine (PVM). The following sections describe and evaluate the prototype.

6.1 PVM

PVM (version 3.3.11) is a MPS that is very simple to use. It supports the most basic message passing mechanisms, such as send, receive, multi-casting and broadcasting. It has none of the more advanced features for example supported by Message Passing Interface (MPI; see [GLS94]), such as advanced communication patterns and virtual network topologies. Instead it supports dynamic task creation and termination, unlike MPI. It is the simplicity and the dynamic nature of PVM that are the reasons for our choice. In the following we will describe the necessary basics of PVM. For a more thorough explanation of PVM, see [GBD+94].

In order to simplify the observation module, we decided to focus on basic communication (send and receive), basic group communication (multi-casting and broadcasting) and dynamic tasks (task creation and termination.) This means that we have not focused on the more exotic features of PVM—we are of the opinion that the subset of PVM we have chosen poses no loss of generality of the case.

6.1.1 Concepts

The PVM makes a network of computers look like one virtual machine. The executing entity in PVM is the task, which is associated with a unique identity, the task identity (also called the tid), and the identity of the parent (the task of which it were created.) Tasks are created dynamically at runtime and can terminate as well at runtime. Each task can dynamically join and leave broadcast groups, which are identified by a name (a string). Broadcast groups are used in the broadcast primitives—instead of explicitly sending a message to well-known receivers, a broadcast message is sent to the current members of a broadcast group.

Each message is associated with the tid of the sender and an integer (the tag) which can be used freely to type messages. The tag is added to the message by the send primitives. In the receive primitives

it is possible to wild-card both sender tid and message tag in order to make more or less selective communication.

6.1.2 Implementation

Making a network of computers (which we define as a group of connected nodes) appear as one virtual machine is handled through the use PVM daemons.

Every node included as part of the virtual machine has a corresponding PVM daemon running. These daemons are responsible for intercepting communication between tasks. When a task sends a message to another task, the message is directed to the PVM daemon on the node of the sending task. If the receiving task is present on another node the PVM daemon directs the message to the daemon on receiving node. If the tasks are present on the same node, the message is not directed to other nodes but directly to the receiving task. Each task is not aware of the location of other tasks on the virtual machine. This is handled by the PVM daemons.

PVM has a dynamic attitude toward the concept of a task. The number of nodes on the parallel virtual machine is not statically defined. Nodes can be added and removed at runtime if needed.

6.2 Model

In order to take advantage of the modeling features provided by the trace storage module, see chapter 5, we have designed a model for a selected subset of PVM primitives to be instrumented with event generating statements. The selected subset is given by the following primitives:

- *pvm_spawn* (for task creation)
- *pvm_send* (for sending point-to-point)
- *pvm_bcast* (for broadcasting to a broadcast group)
- *pvm_mcast* (for multicasting to explicitly specified tasks)
- *pvm_recv* (for receiving)

To each of these we associate an event. In order to model the beginning of a task execution we have added an additional event called PVMANTISPAWN. This event is emitted at the beginning of task execution. These events are described in the classes seen with their properties in figure 6.1. The class PVMMSG is the superclass of all events corresponding to sending or receiving primitives. In figure 6.2 the class PVMTASK is shown.

Besides the normal causal relation between internal events in a task, the only causal relations between tasks are:

- A PVMSPAWN can be immediate predecessor to a finite number of PVMANTISPAWN's.
- A PVMSEND can be immediate predecessor to at most one PVMRECV.
- A PVMMCAST can be immediate predecessor to a finite number of PVMRECV's.
- A PVMBCAST has the same properties as PVMMCAST.

This gives the necessary foundation for creating the causal graph.

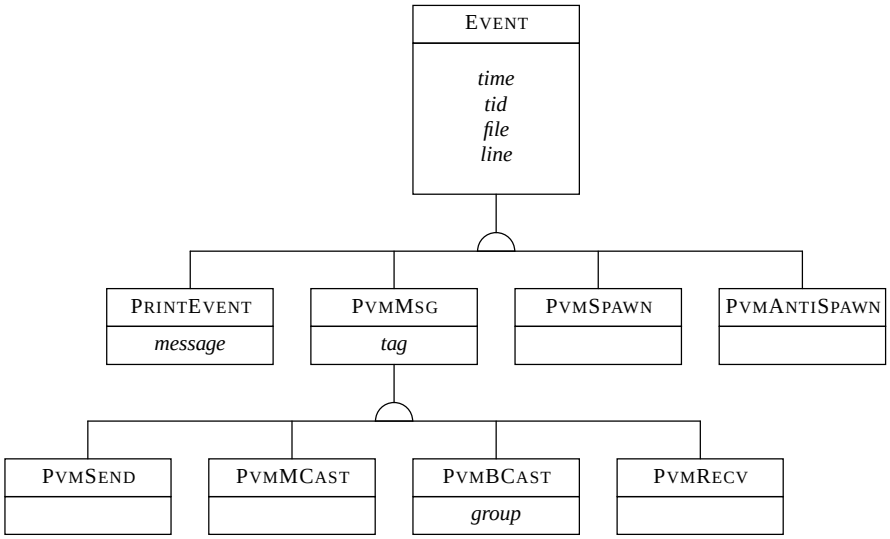


Figure 6.1: The specialization hierarchy of the event classes for the PVM front-end.

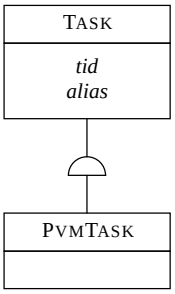


Figure 6.2: The specialization hierarchy of the task class for the PVM front-end.

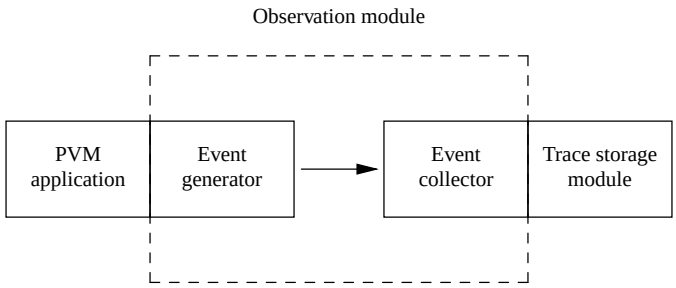


Figure 6.3: This is the architecture of the PVM observation module. The application is linked with an event generation library. Events are sent via PVM to an event collection module, that uses the trace storage module for building the trace.

6.3 The observation module

The architecture of the PVM observation module, seen in figure 6.3, is described in the following sections. All the selected primitives are instrumented in the PVM application and through the event generator, events are sent to the event collector. The event collector receives events and builds the causal graph through the use of the standard interface of the trace storage module. When a PVM application has finished it's computation, the event collector exits.

In the following sections we will describe both peers in the event communication.

6.3.1 Event generation

The goal for event generation is to secure that a minimum of change is needed in the application source for generating events. I order to achieve this we use wrappers that encapsulate the PVM primitive and add additional functionality.

The event generator has two parts: The event generating primitives and the wrappers for the chosen PVM primitives. For each of the chosen PVM-primitives, a C macro has been created that fetches the necessary information from the PVM application and calls an event generating primitive. The corresponding event generating primitive packs the information in a PVM-message and sends it to the event collector.

Two primitives have been added that initializes the trace system: *ti_initial_init* and *ti_init*. The first primitive is called by the very first task of the application. This primitive spawns the event collector. The other primitive is called by all other tasks when they start. In this way a link between the tasks and the event collector is generated. The wrapper for *pvm_spawn* communicates to the newly spawned tasks the tid of the event collector. This information is received by the tasks in the *ti_init* primitive, which also handles the emission of the PVMANTISPAWN.

In addition to this, a *pvm_print* primitive has been added. When this is called with a string as argument, the string is encapsulated in a PRINTEVENT event and sent to the event collector. This will support static debugging.

The wrappers, trace generating primitives and new primitives are supplied in a library—libtipvm—and a C header file. When a programmer wants to enter a debugging session, all needed is adding the initialization primitives, recompile and re-link the application with libtivm. After this, his application will be instrumented at the appropriate places, and initiate a trace collection when run.

In order to time-stamp all events with Lamport times, it was necessary to add new primitives to the PVM

system. These primitives had to be added within the PVM system, because they require a very intimate knowledge of what is going on in the runtime system. The following primitives were added:

- *pvm_lt_set* sets the logical time.
- *pvm_lt_increase* adds 1 to the logical time.
- *pvm_lt_local* returns the local logical time.
- *pvm_lt_peer* returns the last piggy-backed logical time received.
- *pvm_lt_pack* piggy-backs the logical time on the current outgoing message.
- *pvm_lt_unpack* removes the piggy-backed logical time from the just received message and ensures that the local logical time is set correspondingly.

These primitives are used by the wrappers to increase the logical time at each event emission, and they are used to piggyback all messages with logical time. The algorithm described in section 3.3.1 requires both logical time and task identity to be piggyback, however the task identity is automaticly supplied by PVM.

6.3.2 Event collection

As mentioned earlier, the event collector is started by the first task in the system. The event collector is thus a task in PVM like any other task. In the current implementation there is only one event collector running—it is a central event collector. It can be discussed whether or not it is feasible to distribute the event collector, for instance one per node in the PVM. Ordinarily it is not wise to make a central solution, since all tasks have to communicate with the central event collector each time an event is emitted. In our case though, we do not believe this to be a problem. Since all communication between any tasks always goes through the local PVM daemon, this will act as a buffer. We therefore believe the perturbation is minimal using central event collection.

The centralization of the event collector simplifies the functionality significantly. All the event collector does, is listen for events and build the causal graph. For building the causal graph, a modified version the algorithm described in section 3.3.1 was used. The algorithm was modified not to handle synchronous communication.

6.4 Conclusion

In this chapter we have described the implementation of a prototype that makes use of the theories developed in the previous chapters for building a causal graph. The prototype has been developed to try out the algorithm for building the causal graph and to test our trace storage module.

The process of creating the prototype was quite simple, due to the advanced functionality in the trace building module. The biggest problem was the loss of extendibility in the PVM system. We actually had to make alterations to the official PVM system in order to support logical time. This means that users of the front-end not only depends on our front-end but also on our patches to the PVM system. It would be interesting to investigate other methods for building a trace which does not require alterations of the actual MPS.

Chapter 7

Analysis module

In this chapter, we will describe the design of the analysis module of the *traceinvader* system. As shown in the architectural description of the system in chapter 4, the analysis module is placed on the trace storage module and handles the analysis of traces provided by this module.

First design goals for the analysis module will be described. These will ensure the fulfillment of the requirements from section 1.4.1. After this, a description of the module will guide the reader through the use of the analysis module and arguments for the design decisions we have made.

In the following the reader should refer to appendix A. In the descriptions of the module, which is a graphical user interface, references will be made to the figures in this appendix. The description of the user interface will be focused on describing the most central design issues.

7.1 Design goals

The analysis module provides the actual analysis facilities for debugging in the *traceinvader* system. It is placed on top of the trace storage module, which provides the tool with access to traces. The primary goal of the tool is to test the hypothesis. This means, that it must handle the problems of global timing and complexity. The global timing problem is handled through the observation module and the trace storage module. The complexity problem is to be handled by visualizations in the analysis module.

In section 3.2, a proposal for a visualization of the causal graph and several important properties of such a visualization was described. The central visualization in the analysis module will be a visualization of the causal graph, and it must have theses properties. We required in section 1.4.1 that several visualizations must be provided. In addition to the graph visualizations, several alternative visualizations therefore will be developed and supported by the analysis module.

Since the analysis module is to be based on visualizations, it is to be controlled through a graphical user interface. In order to make the system ready-at-hand, we require for this interface to be user-friendly. The user interface is designed to be object-oriented—objects of the user interface can be manipulated. This is a powerful mechanism for providing functionality where it is most intuitive, thus making it easier to learn using the system. It can also provide faster access to functionality. The metaphors used in the user interface must be chosen, so that the requirements for user-friendliness are supported.

7.2 The Navigator

The Navigator is the main window of the analysis module, and is shown in figures A.11 and A.2. The Navigator consists of two parts: the graph visualization and the control panel.

7.2.1 The graph visualization

The graph visualization is a historical visualization of a subgraph of the full causal graph. It has the properties described in section 3.2. The subgraph is visualized in a manner similar to Lamport's space-time diagrams (seen in [Lam78])—also known as *message sequence charts*.

A space-time diagram is a two-dimensional diagram, where the horizontal dimension is the task dimension, and the vertical dimension is the time dimension. This represents the causal graph. In the Navigator, the task dimension is represented by tasks place in a task bar placed in the top of the window. The time dimension is represented by the time values shown in the left of the window. As seen in the figures, we align all events happening in a task horizontally under the task. We also align events horizontally based upon the time in which they have occurred.

The cursor represents the present time—the time at which the user currently is examining. Events placed on the cursor happens at the present time. Events above the cursor has happened in the past, and events below happens in the future. The cursor controls the alternative visualizations in the analysis module as it indicates the present time in all visualizations. In this way an alternative visualization can be easily related to the point in time of the history of the application it represents.

7.2.2 Controls

As mentioned earlier, the graph visualizations always show a subgraph of the causal graph. This graph can be larger than the Navigator itself, which means that functions for handling this are needed. Two approaches can be used here: scrolling the graph and scaling the graph.

Scrolling the graph will enable the user to look at it through a viewport, which means that he can focus his attention to a part of the graph. This will not help him gain an overview of the graph. Instead scaling can be used. If the graph is too large, it can be made smaller by scaling the size of nodes, edges and the distance between nodes. The graph can also be made larger, if it is too small to see the details. This, in combination with scrolling, enables the user to set his point of view to the most appropriate in his case.

We supply scrolling through the use of scroll bars placed directly to the left and below the graph visualization. Scaling is provided through the zoom control placed in the control panel. In figure A.12 is illustrated the difference in the levels of abstractions between a visualization of the full graph and only a part thereof.

In addition to the support of scrolling and scaling, we provide the user with functionality to rearrange the tasks in the task-bar. This seems like a minor functionality, but it can be a large help in gaining an overview of the subgraph. This could be the case in a situation, where the distance between two tasks is so large, that it is impossible to see their communication in the Navigator without scaling the graph to a minimum. In such a case, it must be possible to rearrange the tasks, so that the tasks in question are located closer to each-other. In the Navigator we provide this with functionality for swapping and moving tasks in the task-bar.

The movement of the cursor corresponds to a clock ticking. This metaphor is central to the way we have provided cursor control. We want the user to think of the cursor moving as time passing—not simply as altering the present time. The impression of the application actually running in the analysis module

will be provided this way, even though the *traceinvader* is a postmortem debugging tool. The chosen metaphor means, that the cursor can only be moved from one time to an adjacent time. In order to move the cursor more than one time unit, all intermediate time units must be visited.

Cursor control has been provided by the use of a virtual clock ticking. The time of this clock always corresponds to the position in time of the cursor. It can be started with the forward and backward button in the control panel. The forward button starts the clock with increasing time, which is the way conventional clocks ticks. An increasing clock gives the expression of execution. The clock can also be started with decreasing time. This gives the user an impression of the reverse execution of the application being debugged. The stop button stops the clock. It is also possible to start the clock and let it click until a specified time. This is done by entering the time in the goto field. The speed of the clock can be set with the speed control. When the speed control is set to the minimum value, single stepping is enable.

7.2.3 Expanding and reducing the subgraph

In order to provide the functionality defined in section 3.2, the Navigator must provide support for expansion and reduction of the subgraph. Initially the subgraph shown in the graph visualization contains all events having the following properties: The event is the first in a task or the event has no predecessors. The reason for this is explained in section 3.2; it is exactly these events, that causes all other events.

Expanding the subgraph

Based upon the properties mentioned in section 3.2, the expansion function can be defined to be:

> Given an event, expand the subgraph with all events causally related with the event.

The expansion function can be split up into two functions:

1. Given an event, expand the subgraph with all events causing the event.
2. Given an event, expand the subgraph with all events caused by the event.

Providing both of these functions, ensures that the expansion function is provided also. In addition to this, the expansion can be made more selectively.

In the Navigator, the subgraph is expanded by the use of the cursor. The cursor is always in one of two modes—replay and record—which is chosen by the effect selection in the control panel. When the cursor is in replay mode, the cursor moves as mentioned above. When the cursor is in record mode, the subgraph is expanded when the cursor moves.

All expansion in the Navigator is explicit, that is, the user explicitly marks events, that are to be expanded. All marking is done through events placed on the cursor. It is possible to mark all predecessors or successors to an event for expansion, and it is possible to mark a selection of predecessors or successors for expansion. When the cursor moves, the expansion process is as follows:

1. The cursor is moved from the present time—the source time—to an adjacent time—the destination time.
2. When the cursor moves, all events at the destination time that are marked for expansion appears in the subgraph, along with the necessary edges between them and other events in the subgraph.

This type of explicit and stepwise expansion gives the user full control of the expansion process. This also makes the expansion process more fine-grained, in contrast to the definition of the expansion function.

Reducing the subgraph

The reduction function is like the expansion function defined on the properties described in section 3.2:

> Given an event, remove from the subgraph all events causally related with the event.

We also split this function up into two functions, based upon the same arguments as mentioned above:

1. Given an event, remove from the subgraph all events causing the event.
2. Given an event, remove from the subgraph all events caused by the event.

In the Navigator, these reduction functions are applicable directly to all events in the subgraph.

In figure A.13 is shown two Navigators—one with a full causal graph visualized and one, where a combination of expansion and reduction has been used. The example is based on an application, which is an implementation of a hyper-cubic algorithm. The application consists of 8 identical tasks having identical behavior, so in order to gain an overview of the algorithm, it is only necessary to look a single task and its communication with its 3 neighbor tasks. Through the use of expansion and reduction the result shown in the figure was reached.

7.3 Alternative visualizations

In order to provide several visualizations alternative to the graph visualization, further visualizations have been added to the analysis module for supplying the programmer with additional information in the debugging process. Three visualizations have been designed, which will be discussed in the following.

7.3.1 Animation

The animation visualization is a snapshot visualization designed to show the communication between the tasks in the system one snapshot at a time. This differs from the graph visualization, which is a historical visualization. The motivation for this alternative visualization comes from the fact, that the logical time between a source event and the corresponding destination event, can get quite large. Therefore it can—for example—be difficult to determine the sender and receiver of a message by simply looking at the graph visualization.

The animation visualization consists of a number of tasks and arrows, corresponding to the present system behavior. An arrow between two tasks indicates that there is a causal relation from an event in one of the tasks to an event in the other. Both tasks and arrows have several states, which is indicated by colors in the visualization. An example of the visualization is seen in figure A.3. The tasks in the visualization be supported by several different layouts. In the figure, they are laid out in a circle which could represent token-ring network, but tasks could also be ordered using the underlying communication topology of the application. Many applications are built under the assumption of a specific communication topology, e.g. the hyper-cube topology. In the analysis module, two animation visualizations has been provided; one with a circle layout and one with a hyper-cube layout.

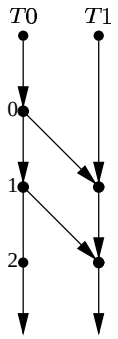


Figure 7.1: The figure illustrates how there can be more than one message active between two tasks at the same time. At time 1, a message is sent, and a message is received.

A task can be in one of two states:

1. The task has an event in the current time (indicated with green.)
2. The task has *no* event in the current time (indicated with red.)

See figure A.3, which shows a snapshot of a broadcast message. The figure also illustrates the use of colors on arrows.

An arrow can convey two facts, depending on its color:

- A source event has occured, but the corresponding destination event has not yet been reached (indicated by yellow.)
- A source event has occured, and the corresponding destination event occurs in the present time (indicated by green.)

The possibility of visualizing the first fact, actually makes the visualization a hybrid between a snapshot and a historical visualization, since the fact is true at all intermediate times between the source and destination event. This is shown in figure A.4: Three slaves have sent their messages, but they have not yet been received. In the present time, the green arrow represents the fact, that the initiator receives the message sent by the last slave. The power of having this history is easily argued; despite this, it is not without downfalls. In order to reliably visualize the history, it necessary to look at more than just the present time. It is necessary to look at the full history of the application, i.e. it necessitates a rather expensive traversal of the causal graph.

Since the visualization shows a history, more than one message can be active between two tasks. The trace on figure 7.1 illustrates the problem. At the time 2, there will be two messages for task number 1. Of course, when drawing this in the visualization, the arrows will overlap. We have chosen to handle this problem by letting source events precede destination events, i.e. green arrows are drawn before yellow arrows. Yellow arrows represents messages not received, which must be the most important property to visualize.

Normally, the user will only need to see the information from the graph visualization. Adding causal relations from the causal graph, that is, causal relations not present in the subgraph, supplements the user with additional information. There are three additional features in the visualization, that help the user to organize the information in the animation. The first is the possibility to select the tasks shown in the visualization. This feature allows the user to minimize the amount of information displayed to a

minimum. The second and third features are the possibilities for rotating and swapping task positions. These allow the user to reorganize the representation to match her vision of the model.

In the following, there will be given three examples on the use of the animation. The first example is to show how the history helps the programmer to determine the context. Figure A.8 shows a sequence of snapshots from the animation, where a group of tasks are responding to a broadcast. The second example is a situation, indicating time passing before messages are received. Looking only at the Navigator, it is difficult to determine the destination of the messages that have been sent. Viewing the causal graph in combination with the animation, it becomes easier to determine the communication patterns. Figure A.10 illustrates the point. The last example shows the hyper-cube animation. The application communicates merely in a hyper-cube topology. Figure A.9 shows a snapshot with hyper-cube layout.

7.3.2 Output

The primary goal of the output visualization is to support static debugging as required in section 1.4.1. This support is realized by providing functionality for connecting static debug information with the graph visualization. The output visualization supplied with the analysis module is shown in figure A.5. It consists of a list of messages from PRINTEVENTs. The output starts in an empty state and messages are then added from the PRINTEVENTS occurring in the present time. Each message is supplemented with task identification, alias and time, hereby making it easy to relate the message to an event.

The ordering of messages in the output visualization follows the time. Messages with equal time are interleaved arbitrarily. All messages that has been output at the current time marked with a red, making it especially easy to relate events occurring in the present time with the messages output. The ordering will be destroyed the moment the user changes the direction of the clock.

In addition to displaying messages in the output visualization, it can be feasible to display the types of events instead. That is, instead of showing messages from specific PRINTEVENTs, the type of *all* events are shown. The idea is to help the user quickly determine the type of the events in the graph visualization. This we support in the output visualization of the analysis module. To the output visualization we have also added indication of each event last in a task. This enables the user to easily determine, that no more events will ever occur in a task, simply by looking at the output visualization.

Figure A.6 shows the output visualization displaying event types. The trace that is being debugged is deadlocked. The reason for the deadlock is that all the tasks have sent a message and now expects to receive one. The problem is that the messages sent, all are addressed to the wrong task, so each of the tasks is expecting a message, but it is never sent. The figure illustrates the solution of this problem, namely the yellow lines indicating the last events.

The indication of the problem is that the last event of all the sorters in the trace is a PVMSEND, which should have been a PVMRECV in this specific case.

7.4 Source

The idea with the source visualization is to fulfill the requirements from section 1.4.1 on source level debugging. It allows the user to connect events directly to the source code. A source visualization shows the path of execution of a specified task. At any time, it shows the place in the source code, that corresponds to an event happening in the current time. The source visualization supplied with the analysis module is shown in figure A.7, where a PVMSEND is highlighted.

By having several source visualizations, one for each task, combined with an animation visualization,

an impression of the global state of the application is given.

The strength in the source visualization lies in the programmers ability to derive further details about a given event than the details given in the trace. As an example the contents of a message could be revealed by seeing the source. This would otherwise only be revealed by the use of static debugging.

7.5 Conclusion

In this chapter we have described how it is possible to structure a debugger around one central visualization, namely the graph visualization. The main window, the Navigator, is used for navigating through a full causal graph. Reduction of complexity is reached through the expansion and reduction functionality, as described in section 3.2.

Alternative visualizations are controlled via the Navigator. This means that at all times it is easier to have a complete overview of the state of an application, merely by combining visualizations. The extra visualizations provided by *traceinvader* are the animation, output, and source visualization. These ensure the fulfillment of the requirements of supporting alternative visualizations, static debugging and source level debugging.

Chapter 8

Conclusion

The motivation for this thesis was formed from a study of the general problems related to debugging and the present technologies for debugging distributed applications. Many tools exist for visualizing the behavior of distributed applications. Several proposals has been made for the use of logical time for debugging but we have only seen few tools actually using logical time. The prototype tool we have developed can contribute to the area of distributed debugging by the combination of logical time and visualization.

The report is based on a study of the general problems related to debugging and the present technologies for debugging distributed applications. This study results in a thesis and a set of requirements for a debugging tool. The study is followed by the theoretical development of methods for fulfilling the requirements stated in the hypothesis. The developed methods are then tested in a prototype implementation of a debugger.

In the following we first relate the result of the project to the hypothesis stated in section 1.4. Then an examination of the *traceinvader* system in a broader perspective is stated, and finally we describe the possibilities for future work.

In section 1.4 a hypothesis was stated for the requirements of a distributed debugger. The goal of the project was then set to proving or rejecting this hypothesis. In the following we will try to evaluate the consequences of following the requirements stated in the hypothesis.

The first requirement stated in the hypothesis was that observing the behavior of the application must be reliable, meaning that the observed behavior must conform to the actual behavior. It was stated that the reliability of the observation is maximized by handling the global timing problem with the use of logical time and ensuring minimal probe-effect.

To test the requirement we have developed a method for observing distributed applications which handles the global timing problem while reducing the probe-effect to a minimum, thus providing a reliable observation of the application. The method is based on a *causal graph* which is build using events time-stamped with Lamport clocks. The foundation for this method was provided through a thorough analysis of algorithms described by Lamport and Fidge.

The second requirement of the hypothesis was that in order to handle the complexity problem it should be easy to gain an overview of the application. Visualizations was suggested as a way to reach this requirement.

Visualizations have been the primary issue for finding a solution to the above. The solution we have developed is an analysis module with an object oriented user interface supporting several different visu-

alizations. The central visualization is an extended space-time visualization. This visualization allows the programmer to show or hide events that influence each other. Additional visualizations are provided for supplying the space-time diagram with extra information. We conclude that the analysis module can be useful for reducing the complexity problem and gaining a reliable overview of application behavior.

Additional requirements for the debugging tool were stated in section 1.4. In the final design these goals are all fulfilled. The challenging requirement was that the tool should not be limited to debugging application from a specific MPS. The debugging tool is designed to support different forms of message passing, but it also supports different MPS's having different information relevant for a debugging purpose.

We find that the overall requirements for the prototype debugger have been fulfilled.

8.1 Perspective

Having argued for the stated requirements being fulfilled, we now examine the use of the prototype debugger in a more broad perspective.

The prototype debugger which we have implemented was developed with the intension of supporting the process of detecting bugs in applications already implemented. This has been the primary goal. However, the use of the debugging is not limited to the debugging process. The tool can with advantage be used as an integrated of a test phase in which specifications for applications are checked. In addition, the debugger can be used to gain understanding of the logic of applications already implemented. Also, the debugging is not limited from being extended with more formal methods for checking the behavior of the distributed applications.

In chapter 1 we decided to focus on distributed applications using message passing for communication. The argument for this decision was based on the notion of distributed application being defined as processes and messages (from equation 1.1.) By introducing a proper level of abstraction the debugger could also be used for other paradigms for communication. This feature can be provided as long as the standard interface in the trace storage module is used for generating a causal graph representing application behavior.

8.2 Future work

The prototype implementation of the debugger has revealed problems in the initial design and given inspiration to improvements. In the following we describe the main goals for improvements.

Extension of graph expansion and hiding facility. In the current implementation of the expansion and hiding features of the space-time visualization only provide basic functionality. This means that using the expansion and hiding features can become quite tedious. By reevaluating and extending the functionality we aim to solve this problem.

Taking advantage of the generality support in the visualizations. The generality support provides a flexible interface both for porting MPS's, but also for the visualizations. In the prototype the default event type and task type hierarchies are kept to a minimum. By refining the hierarchies further with general event and task types, the visualizations can become more detailed depending on the amount of information available.

One of the more abstract ideas arisen during the development of the prototype is the possibility of introducing abstraction in a graph visualization. An example would be to reduce the four events involved

in a client-server communication to a single event. The problem of this idea lies in the description of the abstraction, because of the ready-at-hand requirement.

Cumulative visualizations for supporting the reduction of complexity. Our work has mostly been centered around snapshot and historical visualizations. Cumulative visualizations could provide extra information that help reduce the complexity by supporting the ease of gaining an overview of the application behavior.

Bibliography

[Bat95] Peter C. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Transactions on Computer Systems*, 13(1):1–31, Febuary 1995.

[BHD+95] James Browne, Syed I. Hyder, Jack Dongarra, Keith Moore, and Peter Newton. Visual programming and debugging for parallel computing. *IEEE Parallel and Distributed Technology*, 3(1):75–83, 1995.

[Car96] Neal Douglas Cardwell. Causally-structured output streams for distributed computations. Technical report, College of William & Mary, Virginia, 1996.

[CDK94] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems—Concepts and Design*. Addison-Wesley Publishing Company, second edition, 1994.

[CHS96] Mikkel Christiansen, Jesper Hagen, and Kristian Skov. Codename TRON. The abstract for the author's Masters Thesis, May 1996.

[CM91] Robert Cooper and Keith Marzullo. Consistent detection of global predicates. *SIGPLAN NOTICES*, 26(12):167–174, December 1991.

[DKF93] Suresh K. Damodaran-Kamal and Joan M. Francioni. Nondeterminacy: Testing and debugging in message passing parallel programs. *SIGPLAN NOTICES*, 28(12):118–139, December 1993.

[EP89] Perry A. Emrath and David A. Padua. Automatic detection of nondeterminacy in parallel programs. *SIGPLAN NOTICES*, 24(1):89–99, January 1989.

[FAJ91] Joan M. Francioni, Larry Albright, and Jay Alan Jackson. Debugging parallel programs using sound. *SIGPLAN NOTICES*, 26(12):68–73, December 1991.

[Fid91] Colin Fidge. Logical time in distributed computing systems. *Computer*, 24(8):29–33, August 1991.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidyalingam S. Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, 1994.

[GLS94] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1994.

[HE91] Michael Heath and Jennifer Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, pages 29–39, September 1991. from lecture notes 771 ref 30.

[HMR95a] Michael T. Heath, Allen D. Malony, and Diane T. Rover. Parallel performance visualization: From practice to theory. *IEEE Parallel and Distributed Technology*, 3(4):44–60, 1995.

[HMR95b] Michael T. Heath, Allen D. Malony, and Diane T. Rover. The visual display of parallel performance data. *Computer*, 28(11):21–28, November 1995.

[Hol91] Gerald J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[Jak95] Hank Jakiela. Performance visualization of a distributed system: A case study. *Computer*, 28(11):30–36, November 1995.

[KB95] Thomas Kunz and James P. Black. Using automatic process clustering for design recovery and distributed debugging. *IEEE Transactions on Software Engineering*, 21(6):515–527, 1995.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LM89] Thomas J. LeBlanc and Barton P. Miller. Workshop summary. *SIGPLAN NOTICES*, 24(1):ix–xxi, January 1989.

[Mat89] Friedemann Mattern. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*. Elsevier Science Publishers B.V., 1989.

[MCC+95] Barton P. Miller, Mard D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn parallel performance measurement tool. *Computer*, 28(11):37–46, November 1995.

[Mil91] David L. Mills. Internet time synchronization: The network time protocol. *IEEE Transactions on Communications*, 39(10):1482–1493, October 1991.

[RAM+92] Daniel A. Reed, Ruth A. Aydt, Tara M. Madhyastha, Roger J. Noe, Keith A. Shields, and Bradley W. Schwartz. An overview of the Pablo performance analysis environment. Technical report, Department of Computer Science, University of Illinois, Urbana, Illinois 61801, November 1992.

[Ray88] Michel Raynal. *Distributed Algotithms and Protocols*. John Wiley & Sons, 1988.

[RRZ89] Robert V. Rubin, Larry Rudolph, and Dror Zernik. Debugging parallel programs in parallel. *SIGPLAN NOTICES*, 24(1):216–225, January 1989.

[Tan95] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, 1995.

[Tav94] Luis Francisco Tavera. Three dimensional sound for data representation in a virtual reality environment. Master's thesis, Graduate College of the University of Illinois at Urbana-Champaign, Urbana, Illinois, 1994.

[TU91] Gerald Tomas and Christoph W. Ueberhuber. *Lecture Notes in Computer Science*, volume 771. Springer-Verlag, 1991.

Appendix A

Screen-shots

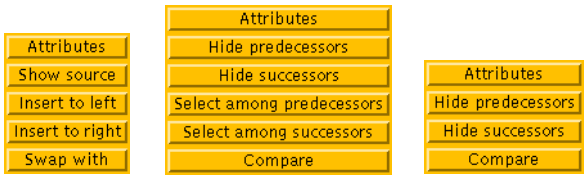


Figure A.1: The pop-up menus of the system. The first is the task pop-up menu, the middle is the event pop-up menu as it appears over the cursor, and the last is the event pop-up menu as it appears outside the cursor.

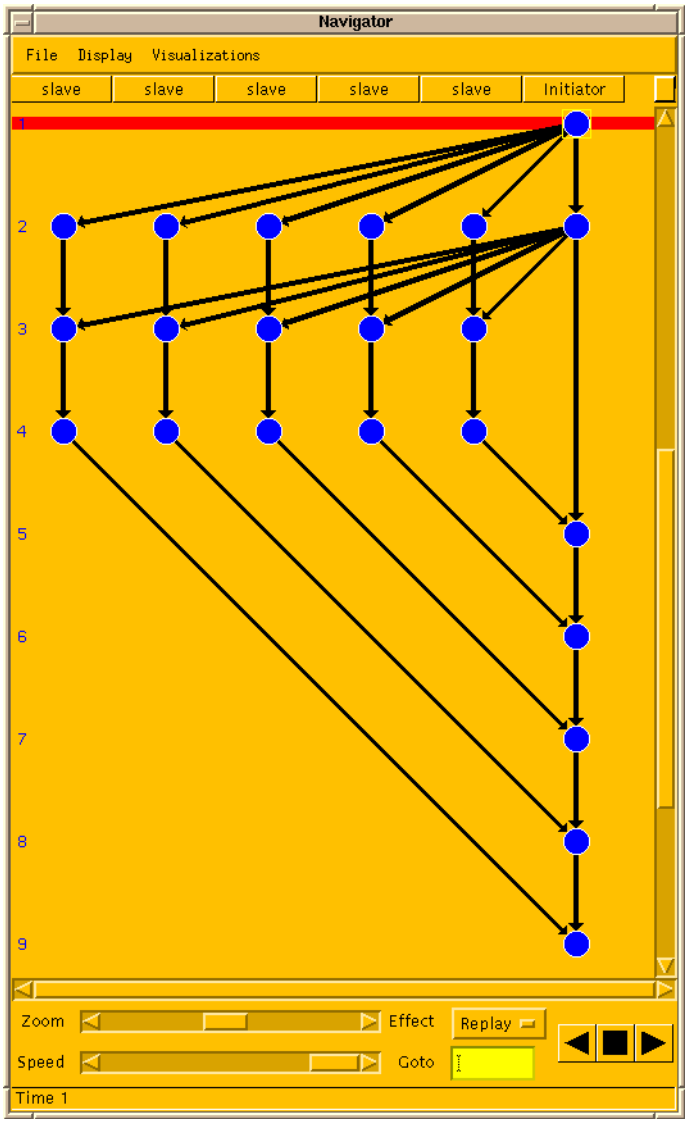


Figure A.2: The Navigator. This window is the main window of the analyzer. At the top is shown the task-bar, then comes the graph visualization. In the bottom is shown the control-panel.

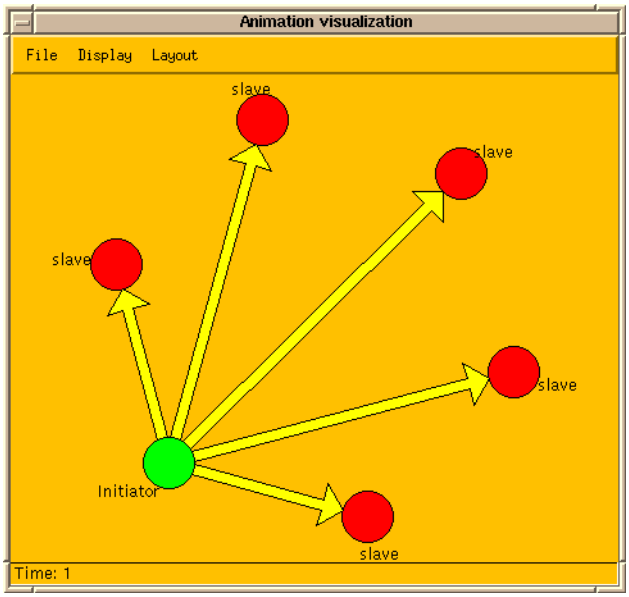


Figure A.3: The circle animation. This shows the snapshot at time 1 in the trace presented in figure A.2.

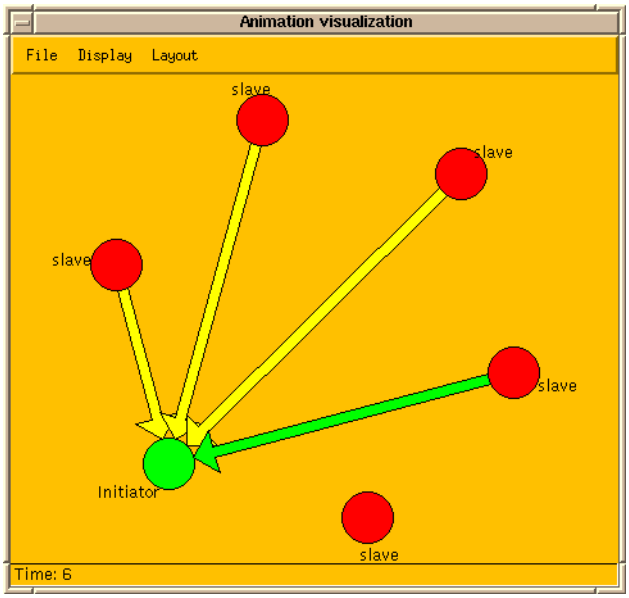


Figure A.4: The circle animation. This is at time 6 in the trace in figure A.2.

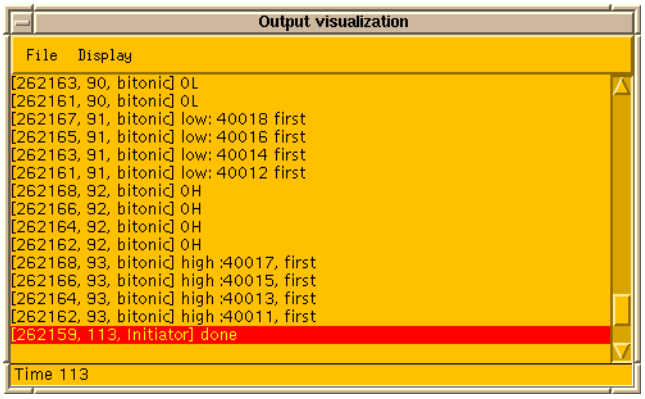


Figure A.5: The output visualization. The red area corresponds to the PRINTEVENT's on the cursor. The output is taken from a bitonic sort example.

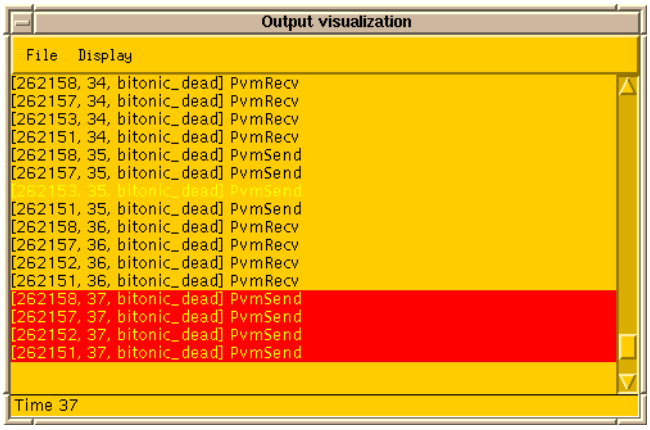


Figure A.6: The output visualization in the mode, which shows the type of all events. This shows also how events that are last in a task are highlighted (with yellow). The output is taken from a dead-locked bitonic sort example.

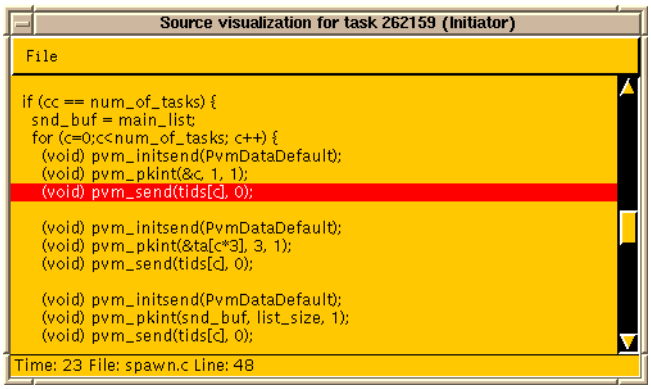
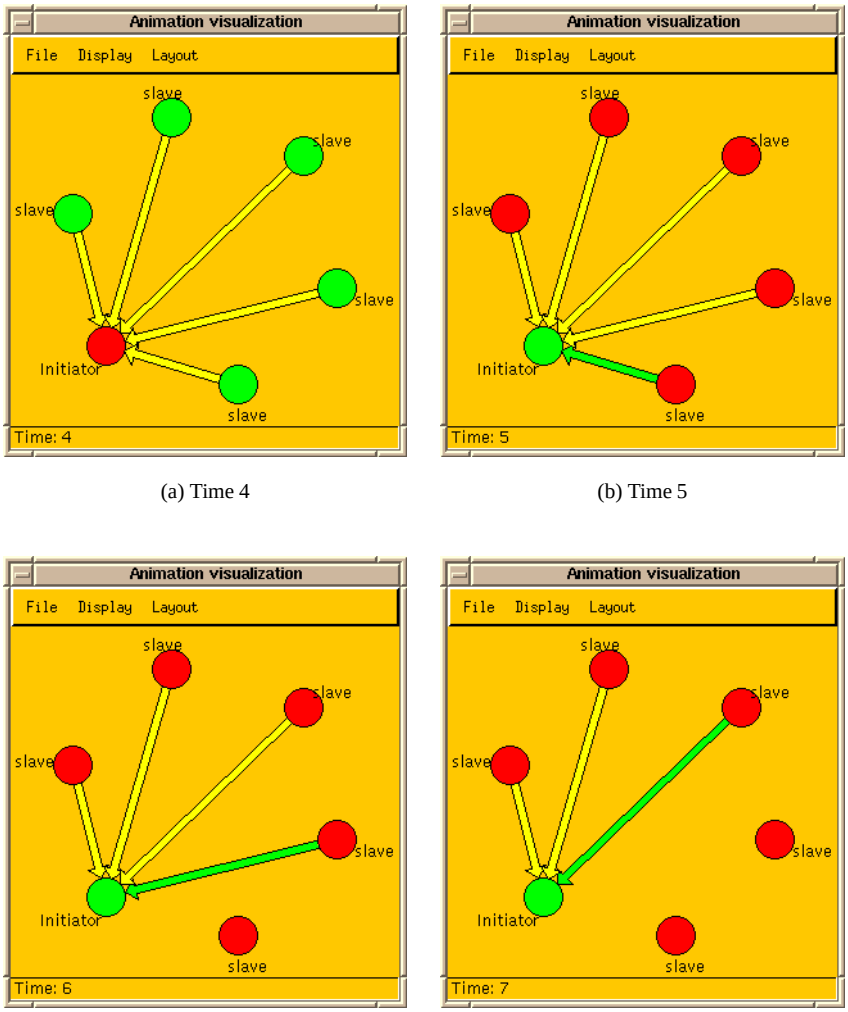


Figure A.7: The source visualization shows the source code for a selected task. The highlighted part indicates the line responsible for an event present on the cursor.



(a) Time 4

(b) Time 5

(c) Time 6

(d) Time 7

Figure A.8: This sequence of snapshots shows how the history in the animation gives a perspective of time. The sequence show how each slave has sent a message to the initiator

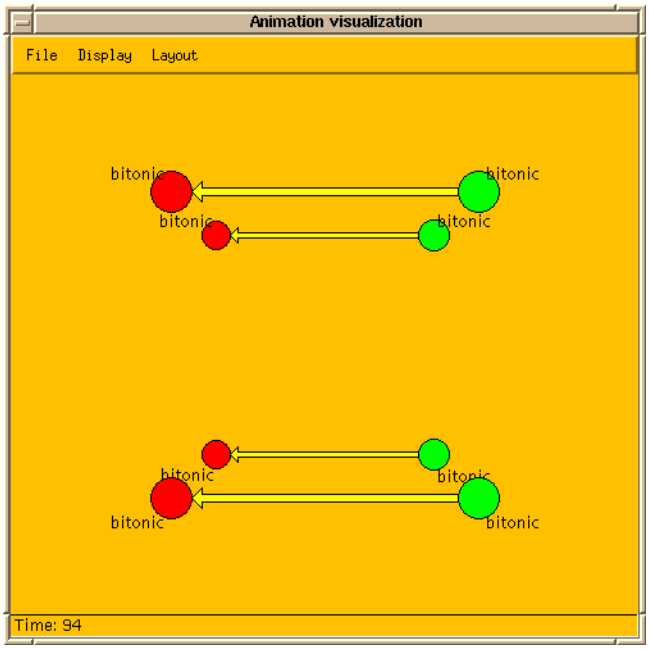


Figure A.9: The hyper-cube animation used on a bitonic sort algorithm.

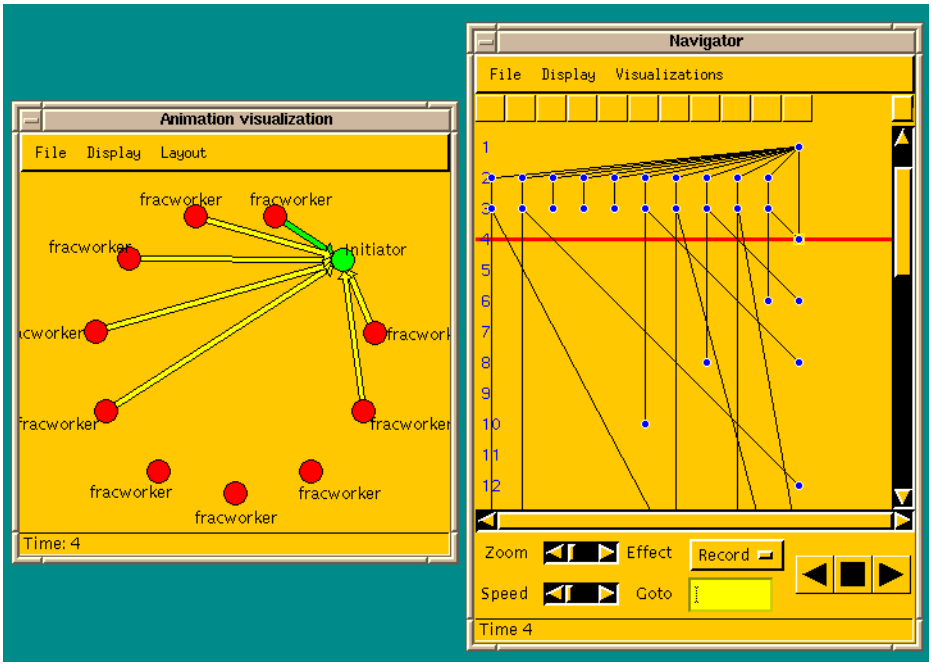


Figure A.10: The figure shows how the animation can make the communication paths easier to determine.

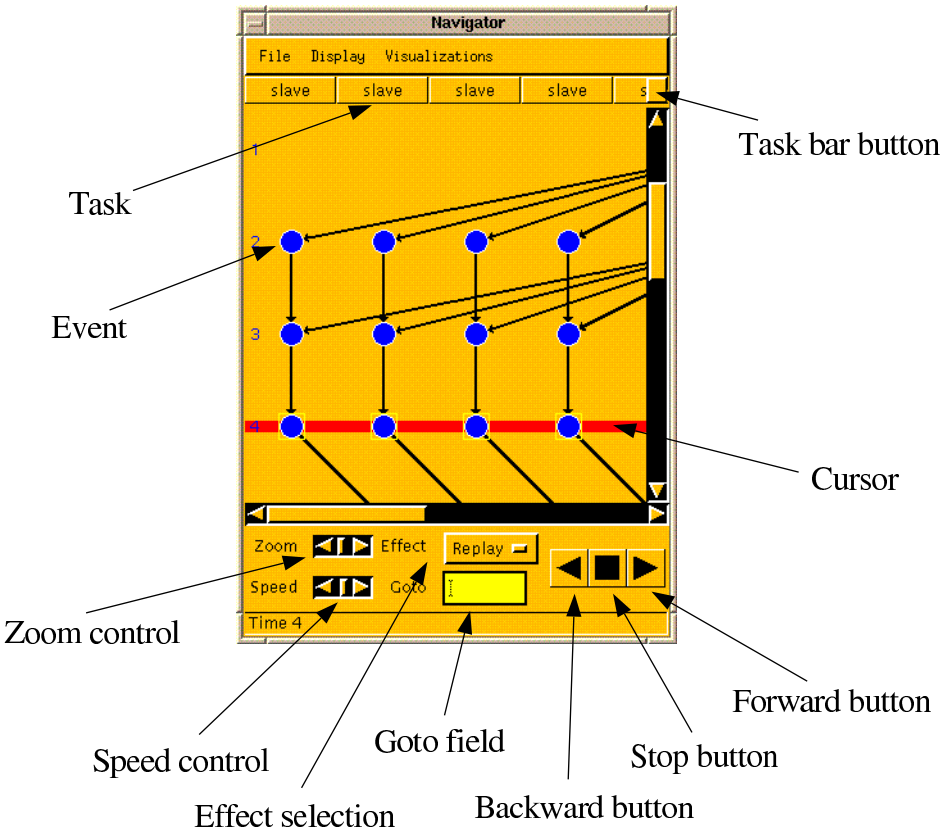


Figure A.11: The Navigator components. The upper part of the window shows the graph visualization, and the lower part the control panel.

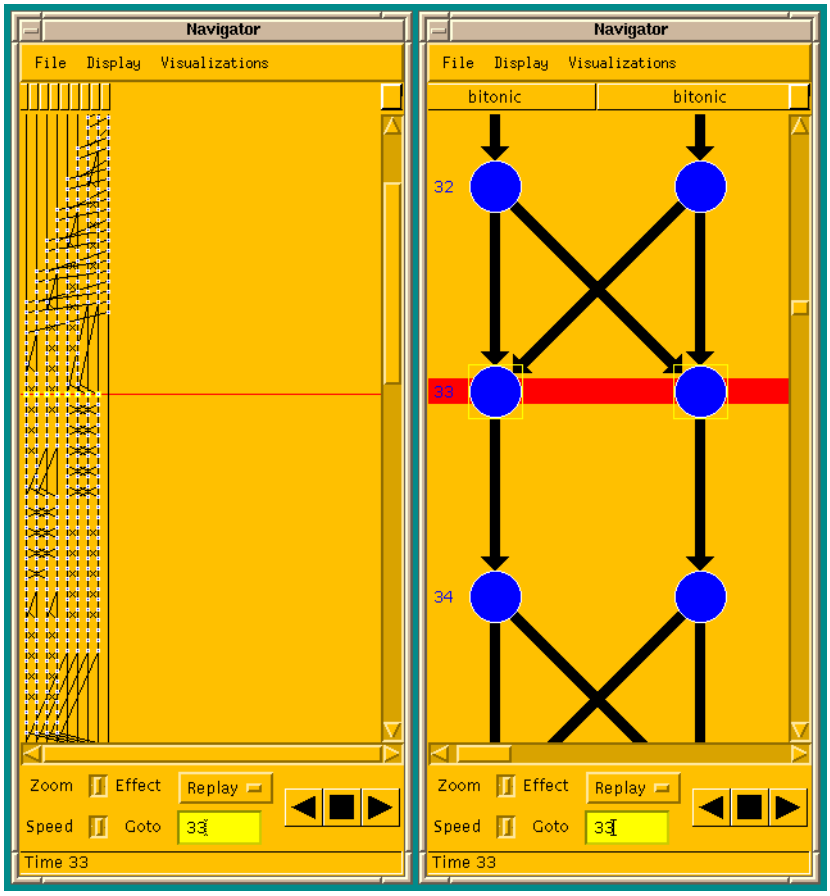


Figure A.12: The left Navigator shows a full visualization in the smallest scale, while the right Navigator shows a part of the visualization in the largest scale.

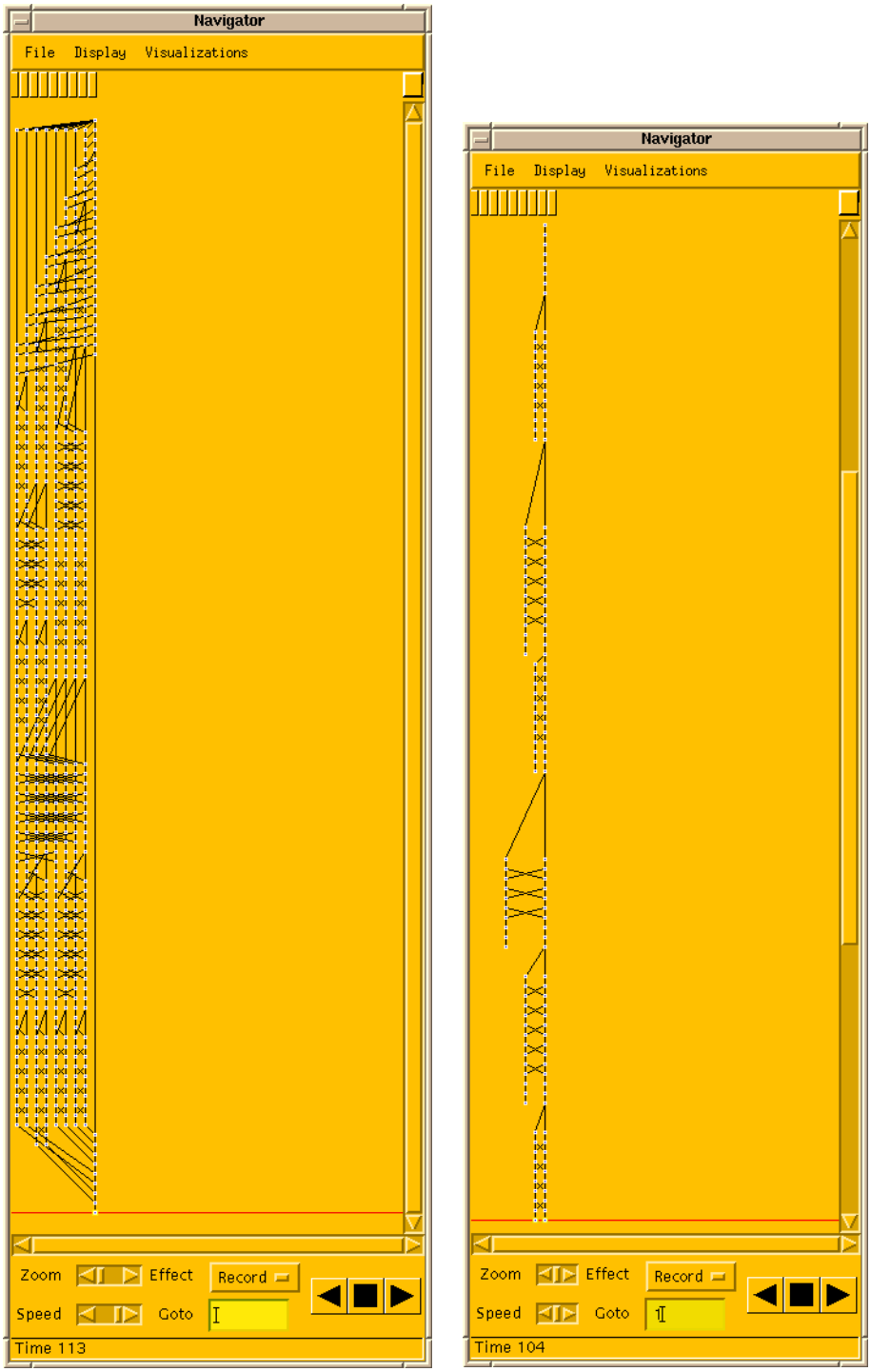


Figure A.13: The left Navigator shows the full visualization, while the right Navigator shows a reduced visualization. In the right Navigator, the focus is on a single task (the one to the left of the rightmost task) and its interaction with other tasks.