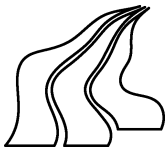
Wot—a language for behavioral abstractions in causality graphs

Mikkel Christiansen Jesper Langfeldt Hagen
Kristian Qvistgaard Skov

June 10, 1997

Aalborg University
INSTITUTE 8 - Institute for Electronic Systems

Department of Computer Science



Title:
Wot—a language for behavioral abstractions in causality graphs.

Theme:
Debugging distributed applications.

Time period:
February 1, 1997–June 10, 1997

Semester:
DAT6/F10D

Project group:
E1-104b

Participants:
Mikkel Christiansen
Jesper Langfeldt Hagen
Kristian Qvistgaard Skov

Supervisor:
Arne Skou

Number of prints: 9

Pages: 73

Abstract:
The goal of this project is the development of tools that can be used for debugging distributed applications. It is based on previous work described in [CHS96, CHS97].

In the previous work we developed techniques for representing the behavior of distributed applications by the use of causality graphs. Graphical visualizations and techniques for placing different views on visualizations were used in order to help over-viewing the behavior of applications.

This project represents a new approach to debugging with causality graphs—namely that of behavioral abstraction. We have developed techniques for placing abstractions on top of causality graphs. These abstractions are linguistically defined in the language Wot, which is the central contribution of this project.

A full formal semantics for Wot is given and a prototype of a subset with small extensions has been implemented. This prototype has been developed with efficiency as the primary concern. A graphical user interface has also been developed in order to ease the task of evaluating the prototype.

This project is a proof of concept of the use of behavioral abstraction in causality graphs in debugging distributed applications. Well-known theories and techniques have been combined in a new way which we believe can help reduce the complexity of debugging distributed applications.

Preface

This report represents the final part of our masters thesis written within the Distributed Systems group at the Department of Computer Science at Aalborg University. The report is written in the period from February 1, 1997–June 10, 1997 by group E1-104b.

This is the final part of our three part masters thesis which has been developed over a three semester period. The two previous semesters cover

1. *Abstract—[CHS96]*: A study of issues related to detecting bugs in distributed applications.
2. *Masters thesis, part one—[CHS97]*: The TraceInvader project with the goal of developing a tool for debugging distributed applications.

The result of the last semester is part two of our masters thesis. During this last semester we have worked on the TraceInvader II project which is based directly on the TraceInvader project. The goal of the TraceInvader II project has been the further development of techniques for debugging distributed applications.

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, June 10, 1997

Mikkel Christiansen

Jesper Langfeldt Hagen

Kristian Qvistgaard Skov

Contents

Chapter 1

Introduction

This project—TraceInvader II—is based directly on the TraceInvader project documented in [CHS97]. In the TraceInvader project we developed a debugger for distributed applications based on message-passing. In the following we will describe the theoretical framework in which we worked during the project and we will evaluate the results of the project.

Based on the framework and the evaluation of the TraceInvader we will in this and the following chapters define the goals for the TraceInvader II project, which will focus on further extensions of theories relevant for debugging of distributed applications.

We begin by describing the theoretical framework which we have adopted for debugging distributed applications.

1.1 Debugging distributed applications

The task of debugging distributed applications is a task vastly more complex than that of debugging sequential applications. The following will be a repetition of the arguments we presented in [CHS97].

A distributed application consists of a set of communicating tasks. These tasks are distributed physically over a set of processors connected through some communication media.

The behavior of any task is defined by the primitive events occurring in the task. A primitive event is an atomic action—this could be the sending or reception of a message, an assignment or a calculation.

We define debugging of an application to consists of three activities:

1. Observation of behavior.
2. Locating bugs through analysis of the observed behavior.
3. Fixing located bugs.

This of course is a cyclic process which terminates when all bugs are removed. A debugger is a tool supporting the first two activities (observation and analysis.)

In [CHS97] we mentioned the following problems in debugging distributed applications:

Complexity Since a distributed application consists of several communicating tasks, it is more difficult to get an overview of a computation. Events occur in different tasks and tasks

communicate thereby influencing on each-other. Locating a bug can be hard because of the difficulty of tracing a cause to its effect. A bug in one task can be caused by a bug in some other task.

Non-determinism Since distributed tasks execute in parallel, non-determinism can occur in executions. The order in which events occur can be interleaved in time or occur in true concurrency. This means that the behavior of a distributed computation can differ from one execution to another.

Probe-effect In order to observe a distributed application it is necessary to insert *probes* which communicate the necessary information to the observer. The problem is that if probes are not carefully inserted in the application the execution can be effected. This is called the probe-effect[1].

Global timing On every processor there is a local clock which always tells the local time within that processor. In a sequential application, this clock can be used to determine the order in time of events occurring in the application. To do the same in a distributed application, it is necessary to have *global time*. The problem of global timing covers the problem of trying to synchronize local clocks on all processors used during a computation.

We can also group these problems into two main problems:

1. The problem of reliable observation: This covers the problems of probe-effect and global timing. If either of these are present, it can be difficult to have reliable observation.
2. The problem of complexity: Besides the problem of complexity, it is also possible to put the problem of non-determinism in this category. Having different paths of execution from time to time complicates the task of locating bugs significantly.

In the following we will describe how we tried to handle these problems in the TraceInvader project.

1.2 The TraceInvader

As mentioned earlier the result of the TraceInvader project was the TraceInvader debugger for debugging *logical bugs* in distributed applications based on message-passing. The purpose of the TraceInvader project was to come up with ideas to handle the problems of reliable observation and complexity. In the following we will describe and evaluate the TraceInvader. The result of this evaluation will be used for setting the goals of the TraceInvader II project.

1.2.1 Ensuring reliable observation

To ensure reliable observation of distributed applications, global timing and minimal probe-effect must be provided. In the following we will describe how this has been achieved in our previous work.

In the sequential case it is easy to relate events with each other due to mutual exclusion. In a single task all events are totally ordered in time, since events are mutually exclusive. This relation between events is called the *happened-before* relation and was proposed for the first

[1]The *Heisenberg Uncertainty Principle* states, that an observer is not passive but effects the observed by observing it. Even though the principle was developed within the area of quantum-physics, it is closely related to the problem of probe-effect.

time by Leslie Lamport in [Lam78]. He proposed that if event a is a send-event and event b is the corresponding receive-event, then a happens before b. This allows one to reason about all asynchronous communication. Later Colin Fidge proposed in [Fid91] some changes to this in order to handle synchronous communication. The relation is defined as follows:

Definition 1.2.1 (Happened-before relation) *The happened-before relation $\rightarrow$ is defined using the following rules:*

1. *Internal events: If e occurs immediately before f within the same task, then $e \rightarrow f$.*
2. *Asynchronous communication: If e is a send event and f is the corresponding receive event, then $e \rightarrow f$.*
3. *Synchronous communication: If e and f participate in the same synchronous communication, then:*
   (a) *If $g \rightarrow e$, then $g \rightarrow f$.*
   (b) *If $e \rightarrow h$, then $f \rightarrow h$.*
4. *Transitivity: If $e \rightarrow f$ and $f \rightarrow g$, then $e \rightarrow g$.*

We say that e happens before f iff $e \rightarrow f$.

The rules 1, 2 and 4 was proposed by Lamport while rule 3 concerning synchronous communication was proposed by Fidge.

The happened-before relation is also called the *causal relation*, since it is possible to see if an event can cause or effect another simply by checking if an event has happened before the other. If this relation between events is present and reflects the actual execution of a distributed application then reliable observation is ensured by examining the causal relations.

Algorithms for creating the relations proposed by Lamport and Fidge exist and they are described in [CHS97]. There we described Lamport's and Fidge's algorithm.

Fidge proposed the use of vector clocks in his algorithm for computing the causal relation. In a distributed application consisting of n tasks, there are n local clocks. In the algorithm each task holds in addition to its own local time also the most recent knowledge of the local time in the other $n-1$ tasks. That is, each task has associated a vector clock of dimension n and this vector clock is updated in a task whenever there is communication to that task or an internal event is performed. The rules for maintaining vector clocks are described thoroughly in [CHS97, pp. 20–22]. Whenever an event occurs in a task, the event is stamped with the current time vector in that task.

Having time vectors attached to events makes it possible to efficiently check for causality, concurrency and synchrony. Fidge defined the operator $<$ on time vectors. $C(e)$ indicates the vector time stamped on the event e. The $<$ operator has the following relation with the causal relation:

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

Knowing that

$$\neg(e \rightarrow f) \wedge \neg(f \rightarrow e) \Leftrightarrow e \text{ occurs concurrently with } f$$

enables us to check if two events occur concurrently.

In the algorithm it is also ensured that time vectors are the same on events occurring synchronously. Therefore the following property holds:

$$C(e) = C(f) \Leftrightarrow e \text{ occurs synchronously with } f$$

The above description shows the handling of global timing which we have adopted in TraceInvader. Next we describe the handling of probe-effect.

Reduction of probe-effect can be done in several ways. The idea is to come up with a probing technique that introduces the least amount of interference.

The most commonly used probing technique is *instrumentation*. In this technique a statement which corresponds to an event is instrumented—a small event-generating statement is inserted before and/or after the statement. These small event-generating statements collect the necessary information about the execution of the instrumented statement and generates an event on behalf of this information. The insertion of event-generating statements can be done statically (at compile time) or dynamically (at runtime). Instrumentation reduces the probe-effect by providing selective probing—only those statements of interest are instrumented. This greatly reduces the amount of events generated.

Another approach which can be used in addition to instrumentation to reduce the probe effect is that of post-mortem debugging. This covers the activity of debugging an application after it has been run. By doing this, no interference is introduced by the debugger. Instead events have to be collected at runtime by some means—for instance by the use of instrumentation. Of course the debugger looses the ability to control the execution—this has to be weighted with the requirements of reliable observation.

The TraceInvader uses time vectors for implementing the causal relation. The algorithms used for generating time vectors was developed in [CHS97] and differs from the algorithm proposed by Fidge by being more efficient (with less probe-effect). In the TraceInvader we combined the techniques of selective instrumentation and post-mortem debugging. Only communication primitives are instrumented and the generated events are collected at runtime and stored for later analysis. To probe events occurring internally in tasks we have provided functionality for manually generating internal events.

All events generated are characterized by a set of properties which represent the unique identity of events or some other information of interest related to the event. Properties are attached to events when these are generated during execution.

1.2.2 Complexity reduction by abstraction

One way of reducing complexity is by the use of abstraction. In the TraceInvader we designed a layered model of abstraction. This is seen in figure 1.1. The model poses different layers of abstraction on behavior (events). Thus we will call this type of abstraction for behavioral abstraction.

The two lowest levels of abstraction correspond to the raw stream of events occurring in the system. The execution level represents the atomic events occurring in the system, while the trace level right above the execution level introduces a filter. Only events which can be classified as important are present at this level. This enables the user to reduce complexity by removing unnecessary events. In the TraceInvader the filter is provided by the instrumentation as mentioned above—only communication primitives and manually instrumented internal primitives result in events.

At the third level of abstraction the causality graph appears. At this level, a structure is posed

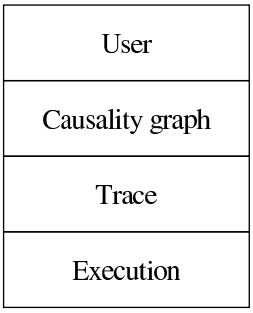


Figure 1.1: The levels of abstraction in the TraceInvader.

upon the events from the trace level. This structure is an acyclic digraph representing the causal relation between events. It is defined as follows:

Definition 1.2.2 (Causality graph) *Given a causal relation $\rightarrow$. Then the causality graph for $\rightarrow$ is the digraph with the events of $\rightarrow$ as nodes, and the edges are created using the rules of definition 1.2.1 except the transitivity rule (rule 4):*

> *If $e \rightarrow f$ and this is because of rules 1, 2 or 3, there is an edge from e to f.*

Even though the transitivity rule is not used, it can easily be shown, that the causality graph has the same information as the causal relation. If $e \rightarrow f$ holds because of transitivity, then there is a sequence $e, h_1, \ldots, h_n, f$ such that:

- $e \rightarrow h_1$.
- $h_i \rightarrow h_{i+1}$ for $i = 1, \ldots, n-1$.
- $h_n \rightarrow f$.

This means that—because of rule 2 in definition 1.2.2—there is a path $e, h_1, \ldots, h_n, f$ in the causality graph. The other way around holds by use of similar arguments.

At the causality graph level it is possible to see relations between events. At levels below, this is not possible since no structure has been imposed on events. By having the causal relations between events, the behavior of distributed applications can be examined by following the paths in the causality graph and thereby ignoring unrelated events.

The three lowest levels (event, trace and causality) are shown in comparison in figure 1.2.

The user level is the highest level of abstraction in the TraceInvader. It is a graphical user interface that provides graphical representations of the causality graph and functions for manipulation. This is described in section 1.2.4.

1.2.3 Architecture

The architecture of the TraceInvader is shown as a data-flow chart in figure 1.3. The TraceInvader is a post-mortem debugger, which means that it is run after the execution of the

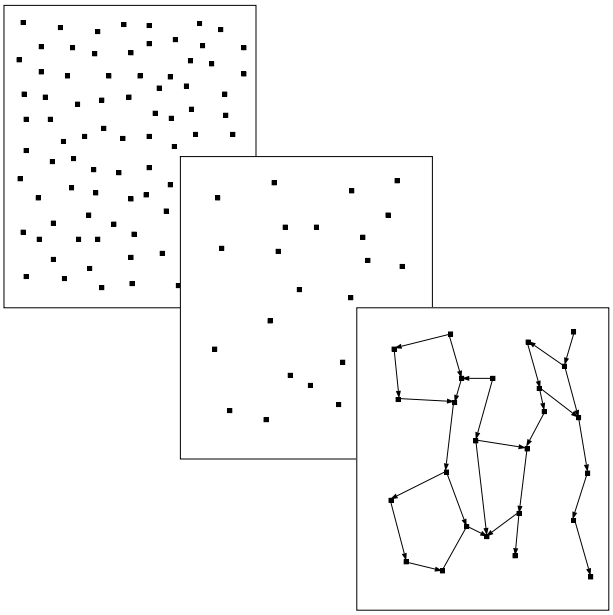


Figure 1.2: This illustrates the differences between the three lowest levels of abstraction. The leftmost box shows the events at the execution level, the middle box shows the events at the trace level and the rightmost box shows the events at the causality graph level.

application being debugged. During the execution, events are generated. These events are present at the trace level of abstraction. Events are collected and structured in a causality graph. This causality graph is then presented to the user through the graphical user interface.

This architecture has a high degree of modularity and is easy to understand due to the simplicity. Due to algorithms developed in [CHS97], it has also proven to be possible to reduce the probe-effect to a minimum while still achieving reliable observation. This is an important property.

1.2.4 Graphical user interface

In the TraceInvader the focus of the GUI was to provide an abstract view of application behavior through a variety of visualizations. In the following we give a short description of the ideas of the visualizations and evaluate the approaches for reducing the complexity in the analysis of causality graphs. Figure 1.4 shows a screen shot of the TraceInvader graphical user interface.

The primary visualization is a causality graph visualization that displays the graph as a space-time diagram. Causality graphs easily grow large, which means that displaying a whole graph will result a complex view of the application behavior. To handle this the primary visualization allows the user to *prune* the graph. This enables the user to specify a reduced view of the application behavior.

Pruning the graph is done through a set of methods provided through the user interface. These methods use the structure of the causality graph to help selecting relevant information. For instance one can select the events that were involved in causing a special event. These methods should help the user to follow certain patterns, hereby ensuring a consistent view of the graph.

Additional visualizations were provided in the TraceInvader. An example is shown to the left in

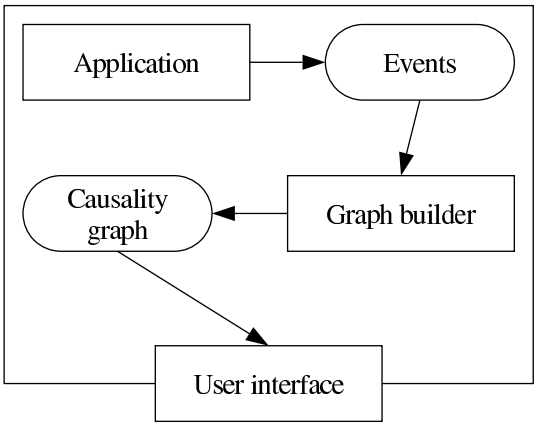


Figure 1.3: The flow of data from the application through the causality graph builder to the user interface is shown here.

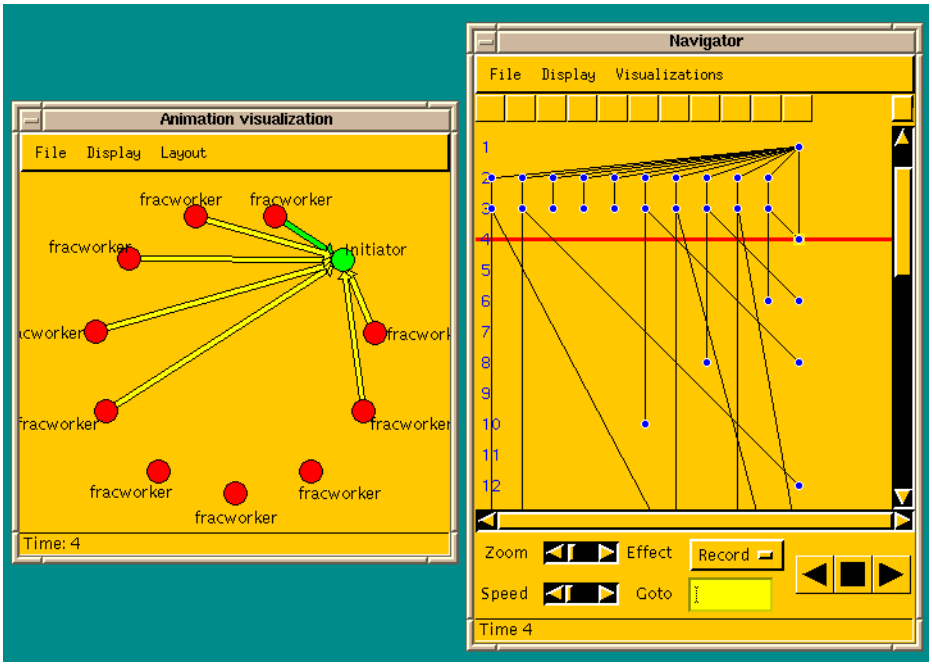


Figure 1.4: A screen shot from the TraceInvader. The navigator window includes complex functionality for displaying causality graphs. Additional visualizations can be used for getting an alternative view of application behavior.

figure 1.4. The goal with these were to support other representations of the application behavior. An example is the animation visualization, which gives an animation of the dynamic behavior of the application. By combining the different visualizations the user can see application behavior from different points of view. This results in a deeper understanding of the behavior and thereby a reduction in complexity.

Two central problems were found that relate to the selected approach for handling the complexity problem. The first problem is the task of pruning a graph to a suitable view. This task can be quite tiresome, because certain types of views require intensive interaction with the user. The second problem concerns the reuse of graph prunings. Having found a bug by pruning a graph, the user will try to fix the bug. After fixing the bug the application is reexecuted, which result in a new causality graph. In order to check whether the bug still occurs, the user might need to reprune the graph to check whether the bug is still present. If the pruning made earlier could be reused, then the user would not need to reprune the graph, and many resources could be saved.

As a consequence of these problems the task of extracting information from causality graphs and hereby examining the behavior of a distributed application still remains complex. The level of abstraction which is provided by the graphical user interface in the TraceInvader enables application behavior to be accurately displayed. However, the functionality for visualization of graphs has proven not to reduce the general problem of complexity. The theory of causality graphs has proven successful; what is needed is another and different level of abstraction on top of causality graphs. This motivates the work presented in this report.

1.3 Conclusion

The TraceInvader project resulted in a debugging tool in which we tried to handle the problems related to debugging distributed applications. These problems were mentioned in this chapter as the problem of reliable observation and the problem of complexity. In the TraceInvader we handled reliable observation by the use of the causal relation, post-mortem debugging and selective instrumentation as probing technique. Complexity is reduced through the use of layers of behavioral abstraction where the complexity is reduced at each layer.

The use of causality graphs has proven important in the analysis of behavior. Central to the TraceInvader is the use of visualizations of which several are provided, all based upon information available in causality graphs. The main visualization on the TraceInvader displays the causality graph and contains functions for manually reducing (pruning) the information displayed. The experiences from using these functions has proven that complexity still remains an important problem.

Based on experiences with the GUI of TraceInvader we state that the purpose of the TraceInvader II project is to provide a different level of abstraction (see figure 1.1) based upon causality graphs. This level is to provide behavioral abstraction which provides reusability and enables automation.

In the following chapters we will investigate techniques for behavioral abstraction and set up the specific goals for this project.

Chapter 2

Behavioral abstraction

In chapter 1 we found that an alternative approach for handling behavioral abstraction was needed. In this chapter we present the EBBA debugging system [Bat95]. This system is interesting because it provides a framework for making hierarchies of events with different abstraction levels.

2.1 EBBA debugging system

The basic idea in the EBBA debugging system is to allow one to locate certain behavioral patterns in a trace. An example could be to locate instances of client-server communications in a trace, hereby allowing one to focus on behavior at a more abstract level. The following will describe the use of abstract models of behavior in the EBBA debugging system [Bat95].

2.1.1 The system

Bates defines *behavior* to be: activity that has observable effects in an executing system. More precisely behavior is defined by the events that are generated during the execution.

The behavior of an application is reflected by instrumenting applications, so that events are generated during execution. The events generated by an application during execution are called primitive events. A primitive event represents non-decomposable fundamental behavior of the application. Examples of primitive events could be *file_open* or *file_read*. The definition of primitive events is done through event classes. An event class defines a type consisting of a name and a set of properties. All event types have the properties time and location which represent the local time of the task and the name of the task. Additional properties describe the actual event, for example the amount of bytes read from a file. The properties are set when the event is instantiated during application execution.

Having defined behavior and events, we turn to the definition of an abstract model of behavior. An abstract model of behavior expresses relationships between different types of events. An example could be a model describing *file_access*, which consist of the events *file_open*, *file_read*, and *file_close*. By introducing such models of behavior it is possible to gain different levels of abstraction when analysing the behavior of applications.

Other applications of abstract models could for instance allow the programmer to search for critical sections in the execution, by describing certain events or communication patterns involved in the critical section. The programmer can also search for faulty models that should

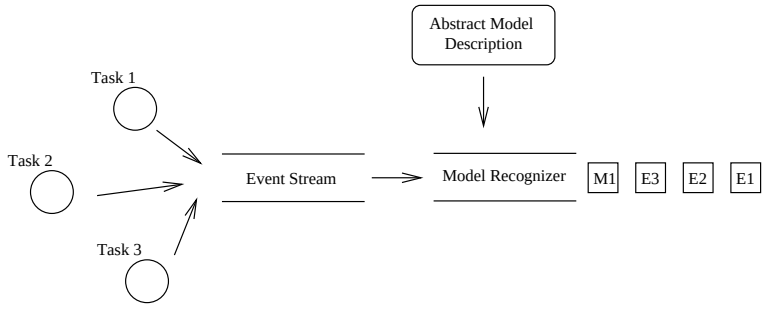


Figure 2.1: The figure illustrates the flow in the EBBA debugger, where the model *M1* was recognized after the event *E3*.

not occur, hereby being able to locate a bug directly.

The abstract models of behavior are used during application execution. Instances of primitive events from the different tasks are sent to an event stream. The event stream is then monitored for occurrences of the user specified models of behavior. If an instance of a model occur, then this information is added to the event stream as a high-level event. This means that different levels of abstraction is introduced directly in the trace. Figure 2.1 illustrates the flow in the EBBA debugger.

2.1.2 Model description

The techniques that the EBBA debugger uses for describing abstract models of behavior are described in this section. Since models represent higher-level events, these also have a name and a set of properties. Furthermore a model has an expression describing the model and a constraining clause restricting the expression.

The language for expressing abstract models of behavior uses a syntax close to the syntax of regular expressions, and has the following primitives for expressing the ordering of events:

Sequential: written $\bullet$; indicates that event instances that match event expression members must follow each other in the event stream, although they are not required to be contiguous.

Choice: written $|$; specifies that an event instance must match any one of the alternative member events.

Concurrency: written $\triangle$, indicates that instances that match its operand events may be arbitrarily interleaved in time. All operands connected with the concurrency operator must match an instance, but their order is unspecified.

Repetition: $+$ or $*$; unary, postfix operators that specify iteration of their operand expression. $*$ will match zero or more instances; $+$ matches a series of one or more instances.

To restrict the expression the model is supplemented with a constraining clause. With the clause it is possible to set restrictions for values of properties of the involved primitive events.

An example model could describe the processing of a file. The expression could be:

$$\mathit{file\_open} \bullet \mathit{file\_read} + \bullet \mathit{file\_close}$$

and the clauses could set restrictions on the filename of the processed file.

More complex models can be described in terms of simple models, hereby using abstraction to describe complex behavior. Each recognized model is instantiated as an event and inserted into the event stream. Models described in terms of simpler models can then utilize these higher level events as if they were primitive events.

The language allows programmers to easily experiment with different models of behavior. This means that the EBBA debugger supports the experimental approach for debugging distributed applications.

Reusability is also supported in the EBBA debugger. Models are described independent of the application being debugged. Therefore the different models can be reused on subsequent executions of the application or on other applications being debugged, as long as they use the same primitive events.

2.2 Conclusion

The EBBA approach to behavioral abstraction is the use of regular expressions describing patterns of events. Patterns can be recognized at runtime thereby resulting in higher-level events describing the patterns. These higher-level events can also participate in patterns, which means that the approach puts no limit on the level of abstraction that can be posed on the execution. This has proven very powerful because it enables a bottom-up approach to the process of building models. This greatly reduces the complexity of model building. The EBBA approach supports reuse of models and automation which we found missing in the TraceInvader project.

In the TraceInvader II project we want to adopt this approach to the layered model of abstraction shown in figure 1.1 in chapter 1. We have already shown the strength of this model. In contrast to the EBBA system which poses a simple total order on events, we want to reuse the results we have gained from TraceInvader project. Therefore we propose a layer right above the causality graph layer which uses behavioral abstraction similar to that of EBBA. This layer must support the information in causality graphs as well as the information in events—EBBA only supports the latter.

Chapter 3

Goals

In the previous chapters we have evaluated the results achieved in the first part of our masters thesis. We concluded the fundamental theories of causality graphs and vector time to be powerful, however the general problems of the complexity of such graphs remains.

We have therefore analysed the EBBA system as a guideline or solution to the fundamental problems of reducing the complexity of extracting information from a causality graphs. Based upon the analysis we now describe in detail the specific plans for this project and outline the rest of the report.

3.1 Language for behavioral abstraction.

Chapter 1 emphasised the need for a higher level of abstraction based upon the concept of causality graphs. With the inspiration of EBBA, we choose to support this level of abstraction through the design and implementation of a causality graph based query language.

The basic ideas are summarized through this hypothesis:

1. The complexity of extracting information from a causality graph can be reduced through the use of a simple language for describing abstract models of behavior.

2. Such a simple language can be implemented in an efficient way and support the automatic detection of abstract models of behavior in a causality graph.

Proving this hypothesis will put us a step further towards the handling of the problem of complexity in debugging distributed applications. The first part of the hypothesis will give us the power of making behavioral models and the latter will enable us to provide efficient search for such models. Efficiency is vital to the usability of a tool supporting behavioral abstraction, since debugging is a highly iterative and most commonly an interactive process as well.

With this hypothesis as basis we will design a language for abstract modelling. This language is to have a well defined formal foundation described by a formal semantics for the language. To test the language and enable a reliable evaluation of the language we will implement a prototype application.

3.2 Requirements for language design

In order to have a more precise foundation for designing and implementing a language for behavioral abstraction, we will require the language to have some basic properties. The requirements to the language are:

- *Abstraction*: It should be possible to allow models to be described in terms of simpler models, hereby making it possible to describe arbitrary complex models of behavior.
- *Simplicity*: A programmer should be able to learn to master the language quickly.
- *Regularity*: The language should provide a small set of basic facilities, that can be used for building more elaborate facilities.
- *Generality*: The primary focus of the language should be on describing models in causality graphs. To a reasonable degree we will try not to specialize the language to use in debugging distributed applications.
- *Relation to theory*: The language should be based on reasoning enabled by theories of causality graphs and time vectors.

3.3 Outline of report

Chapter 4 will give a detailed description of the logical programming language *Prolog*.

Chapter 5 describes the design of the Wot query language. This chapter introduces a simple syntax for abstract modelling of behavior. After that the usage of the language is presented through a series of examples.

Chapter 6 gives a semantic description of the Wot query language. Then the capabilities of the designed language will be made explicit.

Chapter 7 tests the capabilities of the Wot language by describing and evaluating a prototype implementation of a slightly modified version of the designed language.

Chapter 4

Prolog

Having stated requirements for a simple language for behavioral abstraction in causality graphs, we begin by analysing a programming paradigm which will serve as an inspiration for the language design which we will present in chapter 5.

Logic programming languages support high level programming. When using a logic programming language the user is focused the expressing the result of a computation not on how this result is achieved. We choose to to adopt this approach.

In the following we will present the logic programming language Prolog with the goal of providing a conceptual framework for the design of a language for behavioral abstraction. We also show how causality graphs can be represented in Prolog. The description of Prolog in this chapter is inspired by [SS86].

4.1 Logical Programming

A logic programming language provides a language for precisely expressing goals, knowledge, and assumptions in logic. Having described ones thoughts in logic, the language provides a framework for structuring and establishing consistency of these thoughts.

Logic programs allows one to focus on expressing ones knowledge in a set of axioms and rules. The axioms and rules are used to describe the objects and their relations in a structural manner. Once the objects and relations have been described, it is possible check the consistency of the description. The consistency is checked by applying relevant queries to the description. Rules are built from a number of goal statements, which are computed separately, hereby giving a constructive proof of the rule. The Horn clause describes the form of a rule:

$$A \text{ if } B_1 \text{ and } B_2 \text{ and } \ldots B_n \tag{4.1}$$

which should be read A is true if $B_1 \ldots B_n$ are all true. The clause says that a number of goals should hold in order for the A statement to hold. The constructive proof of the Horn clause lies in the ability to read and execute the clause as a procedure of a recursive language. To solve (execute) A, solve (execute) B_1 and B_2 and $\ldots$ and B_n. The Horn clause is important to logic languages, because it gives the foundation for interpreting axioms and rules.

In the following sections we describe the different statements for expressing knowledge and querying on the representation.

4.2 Facts

A *fact* is the simplest kind of statement. It is used to describe the relationships between objects. An example statement is:

father(abraham,isaac).

This fact describes the relation father, that lies between the two objects Abraham and Isaac, meaning that Abraham is the father of Isaac. The two entries in the relation, e.g. Abraham and Isaac are called *atoms*, and the relationship father is a *predicate*. A Prolog program is defined by a finite number of facts.

In the following sections we will exemplify statements through the use of the example Prolog program in figure 4.1. The example program is taken from [SS86].

father(terach,abraham).	*male(terach).*
father(terach,nachor).	*male(abraham).*
father(terach,haran).	*male(nachor).*
father(abraham,isaac).	*male(haran).*
father(haran,lot).	*male(isaac).*
father(haran,yiscah).	*male(lot).*
mother(sarah,isaac).	*female(sarah).*
	female(milcah).
	female(yiscah).

Figure 4.1: A Biblical family database.

4.3 Queries

A query in a Prolog program is used for retrieving information from the description, or database. The basic query is a question of whether a specific relation holds between objects. An example query is: *father(haran,lot)?*, that asks whether or not *Haran is the father of Lot.* According to figure 4.1 this is *true.*

Syntactically queries and facts look the same, except that facts are terminated with a period and queries are terminated with a question mark. An entity with neither period nor question mark is called a *goal.* The fact $P.$ states that P is true. A query $P?$ asks whether the goal P is true.

An answer to a query is retrieved by determining logical consequence. This method uses a set of deduction rules for determining an answer. We have just seen an example where the identity deduction rule has been applied: *from P deduce P.* The query whether Haran is the father is of Lot is a logical consequence of the identical fact that says that Haran is the father on Lot.

In the following sections we will discuss other ways of formulating queries in Prolog.

4.4 Existential queries

An *existential query*, is a query where variables are used for expressing uncertainty of the variables value. In the following we describe existential queries and variables.

A variable in a query stands for an unspecified individual, with a single identity. An example of an existential query is the query: "Does there exist children such that the father is Terach?" which is expressed as: *father(terach,X)?*. In the evaluation of the query, the variable X will be *substituted* with all possible values of X. In this case the values $\{abraham, nachor, haran\}$. It is said that the value X is existentially quantified. For each of the possible substitutions of X a new query is formed. If the new query is true then a result of the query is found. It is a possibility that there are no values to substitute X with, and the query will return false.

The facts found from a query using variables are called *instances*. An example of an instance could be the fact: *father(terach, haran).* which is an instance of the query *father(terach,X)?*, where X is substituted with *haren*.

The deduction rule used for evaluating existential queries is called *generalization*. An example of a generalization is that the fact *father(terach, haran).* implies that there exists an X such that *father(teranch, X)?* is true, namely $X = haran$.

4.5 Universal facts

A universal fact is a fact that use variables for summarising many facts. For instance, if we want to give all the facts for 0 multiplied with a number in Z, we could never list all the facts. By using a universal fact we can write: *times*$(0, X, 0)$., which says that 0 times X is always 0. The variable X is implicitly universally quantified, meaning that it can take any value.

When evaluating a universal fact a rule of deduction called instantiation is used. Instantiation says that: from a universally quantified fact one can deduce any instance of it. This means that one only has to find a fact for which the query is an instance. For example the query *times(0,100,0)?* is true, based on the fact *times*$(0, X, 0)$.

4.6 Conjunctive queries and shared variables

In this section conjunctive queries and shared variables will be described. A conjunctive query is a query that consists of a conjunction of goals. An example is the query: *mother*(X, *isaac*) $\wedge$ *father*$(Y,$ *isaac*)?. The query is true if both goals in the query are fulfilled.

When specifying a conjunction query, it is possible to use shared variables. Shared variables are variables that are shared between the goals of the query. An example is the query: *father*$(haran, X)$, *male*(X)?. The scope of the variable in a conjunctive query is the whole conjunction. This means that a query $P(X), Q(X)$? should be read: Is there an X such that both $P(X)$ and $Q(X)$ holds? Shared variables is a strong tool for restricting the query by restricting the range of the variable.

Searching for solutions to conjunctive queries comes naturally. One must find a solution to each goal of the query, where the shared variables have the same value.

4.7 Rules

The rule statement gives the possibility to define new relations from existing relations. Rule statements in Prolog are written on the form:

Definition 4.7.1

$$A \leftarrow B_1, B_2, \ldots, B_n$$

where $n > 0$. A is the head of the rule, and the B_i where $1 \leq i \leq n$ define the body of the rule. Both A *and the B_i are goals.*

A rule in Prolog is on the form of a Horn clause. Rules and facts are all Horn clauses. This gives Prolog the strength of the Horn clause, e.g. the ability to be interpreted. An example rule could describe the *son* relationship:

$$son(X, Y) \leftarrow father(Y, X), male(X).$$

There are two ways to view a rule in Prolog. The first is to see the rule as an abstraction over a complex query. An example is the query *son(Y, abraham)?*, which is an abstraction over the conjunction in the rule. This way of viewing queries is called *procedural reading*, because they are read like a procedure: "To answer the query: is Y the son of Abraham? answer the conjunctive query is Abraham the father of X and is X male."

An other way of viewing a rule statement is when the $\leftarrow$ is used to denote logical implication. This gives a different way to read the *son*-rule: "For all X and Y, X is a son of Y if Y is the father of X and X is male." This way of reading a rule is called *declarative reading*. Using the rule statement in terms of logical implication, one can describe new relations in terms of existing relations. This allows one to describe arbitrary complex relations, because one can abstract from the details of sub relations.

The *son* rule that was given above is incomplete, since it does not state that a mother can have a son. By adding a rule to the program we can complete the specification of the *son* relationship:

$$son(X, Y) \leftarrow mother(Y, X), male(X).$$

A collection of rules with the same predicate in the head is called a *procedure*. When describing a complex relations ship in a procedure, one can describe the relationship in terms of simpler relationships. Once a complex relationship is described, one can abstract from the different details of the relationship, allowing one to concentrate on describing other relationships.

4.8 Example

This section gives an example of describing abstract models of behavior using Prolog. The idea is to show how suited the logic programming paradigm is for describing abstract models of behavior.

The database shown in figure 4.3 describes a causality graph and figure 4.2 shows the actual graph. The database consists of two different relations, an event and an edge. An event has atoms describing task identity, logical time, and the event type. An event is identified by task identity and logical time. An edge describes an oriented relation between two events, where the first event happened before the second event.

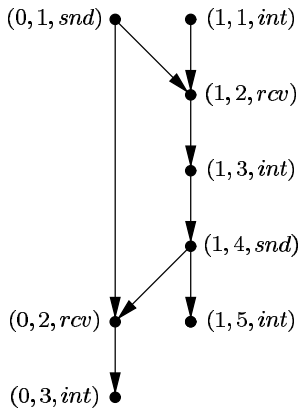


Figure 4.2: A causality graph representing a client-server communication. The information at each event describes the task id, the logical time of the task and type of the event.

```
event(0,1,snd).          event(0,3,rcv).
event(1,1,int).          event(1,2,rcv).
event(1,3,int).          event(1,4,snd).
event(1,5,int).

edge(0,1,0,2).           edge(0,2,0,3).
edge(1,1,1,2).           edge(1,2,1,3).
edge(1,3,1,4).           edge(1,4,1,5).
edge(0,1,1,2).           edge(1,4,0,2).
```

Figure 4.3: The database representing the causality graph.

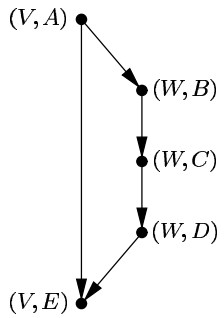


Figure 4.4: Prolog model of simple client-server communication.

In the following we will give two examples of describing abstract models of behavior. The first example is the abstract model for a path, and the second is the abstract model for a simple client-server communication.

The first abstraction is the abstract model for a *path*. If there is a path between two points in the causality graph, then it means that it is possible to come from point (X, A) to point (Y, B) by following the graph. The abstract model for a path is given by the following procedure:

$$\begin{aligned} \mathit{path}(X, A, Y, B) &\leftarrow \mathit{edge}(X, A, Z, C), \mathit{path}(Z, C, Y, B). \\ \mathit{path}(X, A, Y, B) &\leftarrow \mathit{edge}(X, A, Y, B). \end{aligned}$$

The interesting thing with the path procedure, is that the length of the path is unspecified. In order to handle this we use recursion. Having added the path procedure to the database, we can ask queries like: $\mathit{path}(X, A, 2, 5)$?, which find all the possible values of (X, A) from where there is a path to the event labeled with task identity 2 and logical time 5. A problem with the path procedure is that, it is only possible see the start and end point of the path, not the whole path. This can be solved by introducing a *list*, which will be build during the search of a path. Introducing mechanisms for building an instance of the abstract model, require that the programmer focus on this rather that the abstract model. We find this to be inappropriate, since it lowers the level of abstraction.

The second example is a simple client-server communication, where a client requests information from the server. The causality graph from the example include one client server communication. We will try and define a simple abstract model for this communication behavior. First we define *com* representing a message between tasks and then we use this to describe the actual model:

$$\begin{aligned} \mathit{com}(X, A, Y, B) &\leftarrow \mathit{event}(X, A, \mathit{snd}), \mathit{event}(Y, B, \mathit{rcv}), \mathit{edge}(X, A, Y, B), \\ &\quad \mathit{not}\ X = Y. \\ \mathit{client\text{-}server}(V, A, V, E) &\leftarrow \mathit{com}(V, A, W, B), \mathit{edge}(W, B, W, C), \mathit{edge}(W, C, W, D), \\ &\quad \mathit{com}(W, D, V, E), \mathit{edge}(V, A, V, E). \end{aligned}$$

The model is seen on figure 4.4. The model first defines the rule *com*, which defines a communication between two tasks. Using the *com* rule the definition of a client-server communication is given. This displays the ability to use simple abstractions to simplify the description of more complex abstractions.

When querying for all client-server communications in a trace, one will just have to issue the query: $\mathit{client\text{-}server}(E, A, E, B)$?.

4.9 Conclusion

The intension of this chapter has been to provide a conceptual framework for the design of a language for describing behavioral abstraction. With the description of programming in Prolog as inspiration we will in the next chapter design a language for behavioral abstraction.

Compared to the EBBA system described in chapter 2 Prolog provides a higher level of abstraction more suitable for the creation of models of behavior. We wish to adopt the principles of logic programming and create a simple query language for making abstract models. This language is as described to provide a level of abstraction based on top of causality graphs.

An alternative to the design of a new language could be to use the Prolog language directly for abstract modelling. However we need a more abstract and efficient environment for modelling behavior and extracting information from causality graphs.

Chapter 5

Design

In chapter 3 the requirements for a query language describing abstract models of behavior were set. In order to find a suitable method for fulfilling the requirements we studied the Prolog programming language. Prolog has shown to be a high-level language which provide strong abstraction mechanisms.

This chapter presents the design of the Wot query language. The language allows one to describe abstract models of behavior in terms of causality graphs. Causality graphs are characterized by allowing detailed reasoning about causality between nodes in a graph. The Wot language will provide mechanisms for supporting those reasoning possibilities, hereby ensuring that expressiveness of Wot resembles the information available in a causality graphs.

Before we begin it is emphasized that this chapter focus on language design and the capabilities of the language seen from a users point of view. We will not focus on the underlying details which is needed for the language to be constructed. In the chapters to follow we describe how queries are actually evaluated efficiently, but for now we focus on the design of Wot.

In addition we must stress the importance of an understanding of the concept of causality graphs and their structure. This information is essential in order to understand the principles behind the language as the syntax and semantics of Wot is closely related to the structure and information stored in the representation of a causality graph.

5.1 Language concepts

The language constructs of Wot are closely related to the Prolog language. In the following we present the individual building blocks of Wot and show a scheme for using these constructs for specifying abstract models of behavior. Through the description we compare and relate the concepts behind Wot to the Prolog language as this will help in understanding the functionality of Wot.

The previous chapter presented Prolog through concepts such as *facts*, *rules* and *queries*. In Prolog terms the causality graph, which is the basis for Wot, is comparable to a collection of facts. Abstractions in Prolog are specified through the use of rules. This scheme is also adopted in Wot where abstract models of behavior are specified through the rule declarations. Similarly rules in Wot, can be used in queries, as in Prolog. Next we describe the Wot language in detail and clarify the relation to Prolog.

5.1.1 Causality graphs as facts

We begin by describing the relation between causality graphs and facts. Basically we interpret causality graphs as a database of structured facts from which information can be extracted. This resembles the representation of a causality graph that was given in section 4.8. The information included in causality graphs includes properties attached to events, the causal relations between events, and the vector time of the events.

The Wot query language assumes that all information about edges, nodes, properties, and vector time is available. However the causality graph remains the underlying structure and these facts only exits on a conceptual level.

On one specific point the Wot language differs from Prolog concerning facts. In Prolog the facts in the database can be updated dynamically. However in Wot the causality graph, which represents a database of information is statically defined and cannot be updated by Wot queries.

5.1.2 Specifying models of behavior

Having described causality graphs through Prolog facts, we are ready to describe how Wot enables abstract models of behavior to be specified. As described in chapter 4 abstractions in Prolog are specified through the use of rules on the form of Horn clauses. We have decided to provide a similar mechanism for specifying models in Wot. Rules in Wot are specified as:

$$r(args) \Leftarrow E;$$

This declaration binds the name r with the set of formal arguments $args$ to the expression E. Expressions in Wot used for specifying abstract models of behavior. Three types of expressions are included; simple expressions, combined expressions and rule invocations.

Simple expressions

The simple expressions in Wot are closely related to the mechanisms for supporting the reasoning possibilities of causality graphs, described in section 1.2. This ensures that expressiveness of Wot resembles the information available in a causality graphs. The simple expressions are:

Edge expression ($e \rightarrow f$): Models the existence of edges in the causality graph, between nodes e and f.

Sync expression ($e \leftrightarrow f$): Models the existence of synchronous relationship between nodes e and f in the causality graph.

Parallel expression ($e||f$): Models the existence of parallel relationships between nodes e and f in the causality graph.

In these expressions e and f are variables which can be any string.

Three types of expressions are used in a Wot model. These include nodes and edges of causality graphs. Furthermore, expressions expressing comparisons between properties of events are allowed. In order to allow a certain flexibility, these are described through arithmetic expressions:

Arithmetic expression ($A_i\ \theta\ A_j$): Models the existence of relations between properties attached to events.

A_i can be any property ($e.p_i$), a combination of arithmetic expressions ($A_n + A_k$) or a constant expression n. θ can be any of $\{<, \leq, =, \neq, \geq, >\}$.

Combined expressions:

The simple expressions can be combined to model more advanced models than just the existence of edges and nodes. This is done through the use of expressions representing disjunction and conjunctions of expressions.

Disjunction expressions ($E_1 \vee E_2$): Models the disjunction of two expressions.

Conjunction expressions ($E_1 \wedge E_2$): Models the conjunction of two expressions.

These constructs for combining expressions are essential for building abstract models. The scope of a conjunction or a disjunction is the whole conjunction or disjunction. This means that variables used in more than one expression of the conjunction or disjunction are shared between the expressions in which it is used. Shared variables provide a simple technique for binding the expressions of a conjunction or disjunction together.

An example of combining expressions can be illustrated by the following model:

$$e \rightarrow f \wedge (f \rightarrow h \vee f \leftrightarrow g)$$

The expression models the existence of a node e connected to a node f which is related to a node h through an edge or has a synchronous relation with a node g. The repetitive use of f illustrate the use of shared variables for combining expressions.

Rule invocations

Rules are used for building more advanced models of behavior. They are treated equally to simple expressions, meaning that they can be used as simple expressions.

Since rules are treated equally to simple expressions, they can be invoked recursively. Two examples of rule declarations illustrates this property;

$$\textit{a\_rule}(e, f) \Leftarrow e \rightarrow g \wedge \textit{other\_rule}(g, f);$$
$$\textit{path\_rule}(e, f) \Leftarrow e \rightarrow g \wedge \textit{path\_rule}(g, f);$$

The use of model declarations within rules enables complex and advanced models to be specified in Wot. We will illustrate this capability further through a series of examples in section 5.3.

5.1.3 Queries in Wot

Rules and expressions in Wot are in many ways similar to rules and expressions in Prolog. Having described the basic principles behind abstract modeling of behavior, we now turn to the actual queries. The characteristics of queries in Prolog was described in section 4.3. In the following we will explain how Wot adopts the concept of existential queries.

From a syntactical point of view, queries in Wot are simply expressions terminated by a question mark.

A simple query could be: $e \rightarrow f$? This is in Wot interpreted as: "Given the causality graph G, then find all instances of the model $e \rightarrow f$". When this query is evaluated, then all instances of the model $e \rightarrow f$ are instantiated as a new causality graphs with the nodes e and f and an edge connecting the nodes. In Wot terms we say that a query return all possible instances of a model. This is opposed to Prolog that only return a boolean indicating whether there exists a model or not.

More generally the results of expressions evaluated as queries can be summarized as:

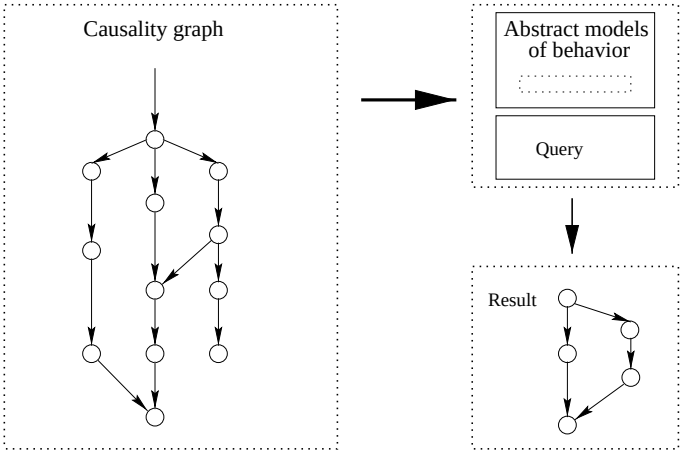


Figure 5.1: Scheme for using Wot language for queries on traces.

$e \rightarrow f$: returns all instances of the model $e \rightarrow f$.

$e \leftrightarrow f$: returns all instances of the model $e \leftrightarrow f$.

$e||f$: returns all instances of the model $e||f$.

$r(args)$: returns all instances of the models described by the rule r.

$E_1 \wedge E_2$: returns all models described by both E_1 and E_2.

$E_1 \vee E_2$: returns all the models described either by E_1 and E_2.

It is worth noting that the order of evaluating the expressions is independent of the result. This simplifies the task of expressing a model, since one does not need to think of the side effects of an expression. Furthermore the evaluation of the query can be optimized by changing the positions of the expressions in the query. We will return to this in chapter 7.

In section 5.3 a set of examples displaying the use of abstract models of behavior and queries will be given.

5.2 Usage of Wot

The basic scheme for making queries in Wot can be seen in figure 5.1. Given a causality graph representing the actual program execution, a query is constructed representing some behavior of interest. To support the requirement for reuse of behavioral abstractions queries can be modelled using rules already defined to make more abstract queries.

The result of a query is a set of subgraphs reflecting the behavioral abstraction specified in the query. If an abstract pattern of behavior is not found, the empty set is returned. Hereby the query language can be used to automatically find instances of subgraphs which is of interest, rather than having to manually detect the specific pattern of behavior.

Actual mechanisms for specifying abstractions, rules, and queries are described after the syntax is introduced in section 5.3.1.

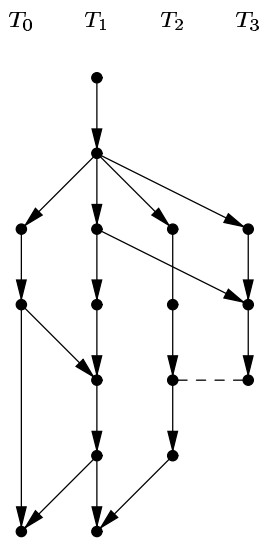


Figure 5.2: An example trace with four processes. The horizontal line symbolizes a synchronization, and not an edge.

5.3 Examples

Having described the main concepts behind the Wot query language, we now introduce the syntax and illustrate the capabilities of Wot for describing behavioral abstractions. This is done by introducing a sample causality graph followed by a series of query examples.

5.3.1 Syntax

To support the simplicity of making queries we have designed a relatively simple language which can be used for modelling arbitrary complex models of behavior. The language includes simple constructs which can be easily learned. These constructs are used for composing queries about application behavior. The syntax is defined as:

$$\begin{aligned} Q &= D_r; E\ ? \\ D_r &= r(x_1, \ldots, x_n) \Leftarrow E; D_r \mid \varepsilon \\ E &= e\ \theta_1\ f \mid A_1\ \theta_2\ A_2 \mid r(x_1, \ldots, x_n) \mid E_1\ \wedge\ E_2 \mid E_1\ \vee\ E_2 \\ A &= n \mid e.p \mid A_1 + A_2 \mid A_1 - A_2 \mid A_1 * A_2 \mid A_1/A_2, \end{aligned}$$

where $\theta_1 \in \{\rightarrow, \leftrightarrow, ||\}$, $\theta_2 \in \{<, \leq, =, \neq, \geq, >\}$, $n \in \mathbf{Num}$ and **Num** is the set of all integer numerals. A query in Wot consists of rule declarations followed by expressions.

5.3.2 Causality graph example

Figure 5.2 shows a simple example of a causality graph. The graph is a causality graph that illustrates the behavior of a distributed application with 4 tasks. Task 1 spawns 3 other tasks and then communicates with task 0 and 3. Task 2 and 3 performs a single synchronization before the they end, this is symbolized with the horizontal line. To use the Wot language for queries it is required that information about properties and time stamps are available when evaluating.

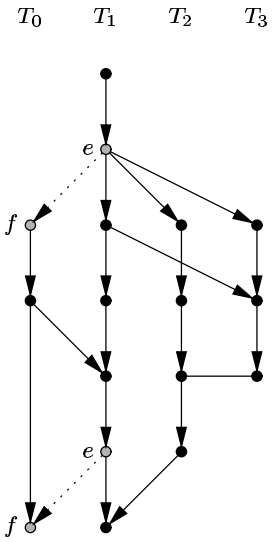


Figure 5.3: First simple example of the result of a query.

In the following examples we will assume this implicitly and focus on the functionality of the syntactical constructs.

Each node has a corresponding set of properties reflecting the information attached to generated events. In the following examples we will for simplicity assume only two properties attached to events. This is task identity *tid* and node identity *type* which can have the value *int*, *snd*, and *rcv*. *int* describe an internal event. On a specific event e the *tid* property is accessed using the notation $e.tid$.

5.3.3 Basic queries

The first example query illustrates the simple need of searching for any communication between two tasks. Let us assume we were interested in all communications from task 1 to task 0. This could be checked with the following example:

Wot-example 5.3.1

$$e \rightarrow f \wedge e.tid = 0 \wedge f.tid = 1?$$

The result is shown in figure 5.3. Evaluating the edge operator will result in a search in the causality graph. During this search all possible instances of the model $e \rightarrow f \wedge e.tid = 0 \wedge f.tid = 1$ are instantiated as a new causality graphs with the nodes e and f and an edge connecting the nodes.

The conditions in this example are the requirements for task identities and this is checked by examining the properties attached to each event ($e.tid = 0 \wedge f.tid = 1$). Figure 5.3 shows two different results which meet the query and reflects bindings of variables $e \wedge f$. The semantics in chapter 6 describes explicitly how variables are bound to events and how subgraphs of causality graphs are returned from queries.

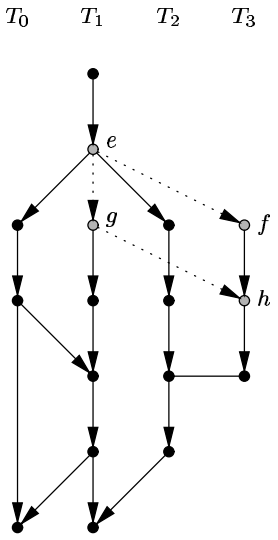


Figure 5.4: Query showing simple use of abstraction.

The example shows how the conjunction operator is used to combine expressions in Wot. If the conditions for task identities were omitted the query would simply return all nodes connected by an edge. However with the simple constructs of Wot the amount of information returned from a query can be effectively minimized.

If we were interested in any synchronous events, a simple $e \leftrightarrow f$ query would return the two events in the example causality graph, which have communicated synchronously. Similarly a $e||f$ query will return all possible pairs of events which has no causal relation.

The motivation for including the parallel and synchronous constructs is based on the possible reasonings about application behavior. The reasoning is directly enabled the by causal ordering of events and the use of vector time. As described in section 1.2 synchronous events are present if they have identical vector times and two events are parallel according to Fidge, if they have no causal relation.

5.3.4 Queries using abstraction

The above example shows how Wot can be used for making simple queries using the basic constructs of the language. The next example illustrates how more complex queries can be constructed by including definitions of rules as abstractions of observed behavior. An example of this is shown here:

Wot-example 5.3.2

$$com(a,b) \Leftarrow a \rightarrow b \wedge b.tid \neq a.tid;$$

$$e.tid = 1 \wedge com(e,f) \wedge e \rightarrow g \wedge$$
$$e.tid = g.tid \wedge com(g,h) \wedge f.tid = h.tid?$$

This example shows a query searching for two successive communications from one specified task ($e.tid = 1$) to any other task. The rules try to bind variables supplied as actual parameters

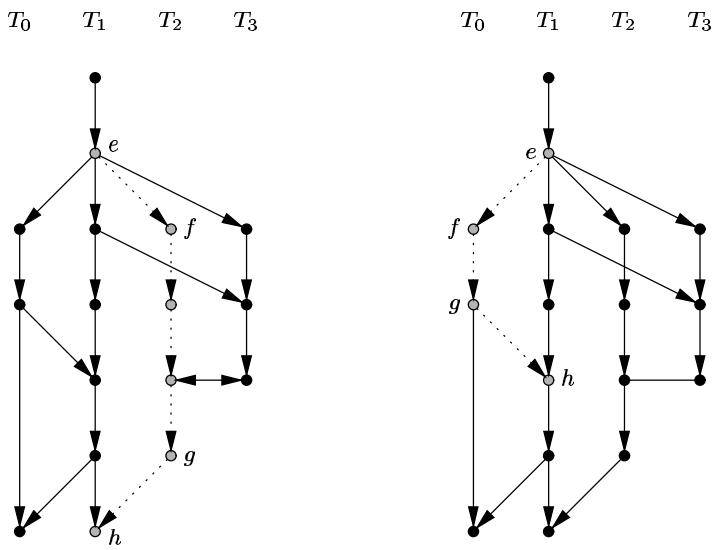


Figure 5.5: Query showing the use of recursion.

by reference. The simple $com(a, b)$ binds variables to nodes which reflects a communication between to tasks; that is two nodes with different task identities (events within a task has the same *tid* value).

The query shows the use of abstraction as the definition of $com(a, b)$ is reused to find successive communications from one task to another.

5.3.5 Queries using recursion

Wot-example 5.3.3

$com(a, b) \Leftarrow a \rightarrow b \wedge b.tid \neq a.tid;$

$mainpath(a, b) \Leftarrow (a \rightarrow b \wedge a.tid = b.tid) \vee$
$(a \rightarrow c \wedge a.tid = c.tid \wedge b.type = int \wedge mainpath(c, b));$

$clientserver(a, b, c, d) \Leftarrow com(a, b) \wedge mainpath(b, c) \wedge com(c, d);$

$e.tid = 1 \wedge clientserver(e, f, g, h)?$

This example shows how queries can be made more dynamic through the use of recursion. This query will detect any client-server pattern with task 1 as client and any other task as server. The query shows an advanced query with rule abstractions serving as a library for behavioral abstractions.

The *mainpath* rule shows the use of recursion which enables any number of internal events ($type = int$) to be performed within the server process before a message is returned to the client task with the identity 0. The *clientserver* rule first searches for a communication from task 1 to any other task. Then the `mainpath` rule is used to search for any number of events in a server task. The *mainpath* is constructed via the $\vee$ expression $(E_1 \vee E_2)$. E_1 ends the recursion if no more events within the same process can be found. E_2 searches the next event

and calls *mainpath* recursively to find the next part of the path.

The use of recursion enables *invariants* to be placed on queries. The example shows this as the *mainpath* rule requires task identity to be the same and the substitution of c to be an internal event before the rule is called recursively again.

The left part of figure 5.5 shows how the result of a query is subgraphs with bindings and extra anonymous nodes which satisfies the query. Anonymous nodes reflect substitutions within the scope of rules. Using the *mainpath* rule can add any number of nodes to a result. The last expression of *clientserver*: $com(c, d)$ ensures that only execution paths which ends with a message back to the client task is returned as a possible result. This again illustrates the important property of the semantics of Wot; queries are evaluated from left to right and a result is returned *only* when all requested bindings are satisfied. This is explicitly described in chapter 6.

5.4 Conclusion

This chapter has described the design of Wot. The language is designed to be simple but with powerful mechanisms for building abstract models of behavior. The language can be used for specifying search patterns in a causality graph. Furthermore the simple expressions of the language provides an expressiveness resembling the information available in a causality graph decorated with vector time.

Abstraction is provided through a simple but powerful mechanism for specifying rules of behavior.

The examples in this chapter have shown the basic functionality of the language. However the Wot example graph used, which was rather simple, did not illustrate the power of Wot used on a large scale graph. This is of course relevant, and we will return to these issues in chapter 7 were we evaluate the language through a prototype implementation.

In the next chapters we make the functionality of Wot explicit through a semantical description and a description of how queries are evaluated. A semantical description can be made the basis for the algorithms for evaluating Wot queries. As mentioned in chapter 3, we require Wot to be implemented in an efficient way. In order see if this is possible, we will design and implement a prototype version of Wot with the explicit goal of efficient evaluation. This prototype will be described and evaluated in chapter 7.

Chapter 6

Formal semantics

Having described the design of the Wot query language, the following chapter continues with a description of the formal semantics of Wot. This chapter serves two purposes. First of all the semantics gives a precise understanding of the functionality and capabilities of Wot. The Wot language uses causality graphs as input and produces subgraphs of these as output and the semantics will clarify precisely the results returned from queries. Secondly the semantics forms an important foundation for the actual evaluation of queries which is described in the next chapter. From the semantics the algorithms for evaluating and searching for subgraphs can be constructed unambiguously.

The functionality of Wot is described using natural semantics [NN92]. We have chosen this type of semantic description as opposed to denotational semantics, as we believe this kind of semantics is easier to understand and lies closer to the actual implementation.

The next sections begin by describing a foundation for the semantics, in which we present the initial definitions. This is followed by a description of two transition systems which represent the actual evaluation of queries.

6.1 Foundation

In this section we begin by presenting a simplification of Wot by exclusion of disjunctions. Following this we present a slightly moderated syntax of Wot which will give a better support for the semantic description. After that we state the definitions necessary for representing causality graphs. A definition for directed acyclic graphs is given followed by a definition of vector time. An environment model is then presented which will be used for representing:

1. Information attached to events: properties and vector time.
2. Information about variable bindings during evaluation.
3. Information about variable declarations.
4. Information about rule declarations.

These definitions will be used in the semantic description of the evaluation of queries.

6.1.1 Exclusion of disjunction

In this section we argue for removing disjunctions from the semantic description of the language. The main argument for this exclusion is to simplify the evaluation of queries. As described in chapter 4 Horn clauses gives a constructive proof of rules that are conjunctions. We wish use a similar approach in Wot by simplifying rules and queries to conjunctions. This enables an evaluation strategy similar to that of Horn clauses.

In order to describe the simplification, we first describe the relation between proposition logic and Wot. An expression in Wot is a proposition in a proposition logic. Variables are assigned to nodes in a causality graph and a proposition is Wot is a proposition of the existence of a given property of the graph. Therefore the associative, commutative, and distributive laws of proposition logic can be applied to expressions in Wot[vD80, pp. 20].

The simplification takes advantage of the fact that by use of the distributive law, any expression with a set of conjunctions and disjunctions can be converted to disjunctive normal form (DNF) through the use of the distributive law.

Definition 6.1.1 (Disjunctive Normal Form) *If F is any expression involving the expressions $E_1, E_2, ..., E_n$, then F is equivalent to an expression in disjunctive normal form:*

$$\bigvee_{i=1}^{k} \bigwedge_{j=1}^{l_i} e_{i,j}$$

where each $e_{i,j}$ is an expression E_k.

This means that all the conjunctions of the expression can be evaluated independent of the disjunctions. Therefore we can exclude disjunctions from the semantics.

An example of a conversion of a Wot query to DNF could be the following expression:

$$E = E_1 \wedge E_2 \wedge (E_3 \vee E_4)$$

When converted to DNF the expression is transformed to:

$$E' = (E_1 \wedge E_2 \wedge E_3) \vee (E_1 \wedge E_2 \wedge E_4)$$

The semantic description presented in this chapter requires that expressions are conjunctions of expressions. As explained this can be achieved by converting rule declarations and queries to DNF. The consequences of this conversion are summarized as:

- For every rule declaration containing disjunctions, the rule is replaced with a set of rules with the same signature. The new rules each represents a conjunction in the converted rule.
- For a query containing disjunctions, the query is replaced with a set of new queries only containing conjunctions. Each new query represent a conjunction in the converted query.

If a query has been converted, then the result of the evaluation is the union of each of the new queries; if we name the evaluation of expression E by the function *eval* the result could be characterized as:

$$eval(E) = eval(E') = eval(E_1 \wedge E_2 \wedge E_3) \cup eval(E_1 \wedge E_2 \wedge E_4)$$

Having argued for the exclusion of disjunctions from the semantic description, we are ready to present a slightly modified syntax.

6.1.2 Syntax

The semantic description is based on a new version of the Wot syntax presented in chapter 5. The syntax used here does not include the disjunction expression, but does include explicit declarations of variables both locally and global. The reason for including explicit declarations of variables is to simplify the semantics. Since the necessary declarations of variables easily could be extracted from a query, this addition to the syntax does not change the functionality of the language.

Definition 6.1.2 (Syntax) *The syntax of the query language Wot is defined to be*

$$\begin{array}{rcl} Q & ::= & D_r; D_v; E? \\ D_r & ::= & r(x_1, \ldots, x_n) \leftarrow D_v; E; D_r \mid \varepsilon \\ D_v & ::= & \texttt{var}\ x; D_v \mid \varepsilon \\ E & ::= & e\ \theta_1\ f \mid A_1\ \theta_2\ A_2 \mid r(x_1, \ldots, x_n) \mid E_1\ \wedge\ E_2 \\ A & ::= & n \mid e.p \mid A_1 + A_2 \mid A_1 - A_2 \mid A_1 * A_2 \mid A_1/A_2, \end{array}$$

where $\theta_1 \in \{\rightarrow, \leftrightarrow, \|\}$, $\theta_2 \in \{<, \leq, =, \neq, \geq, >\}$, $n \in$ **Num** *and* **Num** *is the set of all integer numerals,* $e, f, p, r \in$ **Name**.

Q a query, D_r a rule declaration, D_v a variable declaration, E an expression and A an arithmetic expression.

Q *is the set of all queries.* $\mathbf{D_r}$ *is the set of all rule declarations.* $\mathbf{D_v}$ *is the set of all variable declarations.* **E** *is the set of all expressions.* $\mathbf{E_s}$ *is the set of simple expressions (*$e\ \theta_1\ f$ *and* $e.p_1\ \theta_2\ e.p_2$*).* **A** *is the set of all arithmetic expressions.*

Variables are declared explicitly as described and disjunction expressions are removed from the syntax. Next we continue with the initial definitions which will lead up to the semantics presenting evaluation of queries.

6.1.3 Digraph

Central to the foundation for the formal semantics of Wot is the notion of digraphs, since the language basically works on digraphs. The traditional definition of directed graphs is used:

Definition 6.1.3 (Digraph) *A digraph is a 2-tuple* (N, E)*, where* $N \subseteq$ **Node**, $E \subseteq$ **Node** $\times$ **Node** *and* **Node** $= \mathbb{Z}$*. Given a digraph* $g = (N, E)$*, we define* $N(g) = N$ *and* $E(g) = E$*. N is called the nodes of g and E is called the (directed) edges of g. The set of all digraphs is denoted* **G**.

In the following sections we will extend this definition of digraphs to a level of abstraction more appropriate to the formal semantics of Wot.

6.1.4 Vector time

The theory described in section 1.2.1 concerning the ability to reason about causal relations between events is closely related to the use of vector time. Having vector time attached to events is necessary for efficiently being able to determine if events are causally related, synchronous or parallel. We therefore define vector time:

Definition 6.1.4 (Vector time) *A time vector is a 2-tuple* $(i, t) \in \mathbf{VT} = \mathbb{Z} \times \mathbb{Z}^n$*, where* t *is an* n*-tuple indicating the knowledge of the* n *clocks in a system and* i *is the index of the local clock.* n *is denoted the dimension of the time vector.*

Defined on a time vector are the boolean operators $=$*,* $\rightarrow$ *and* $||$*. Given two vector times* $v_1 = (i_1, (t_{1,1}, \ldots, t_{1,n}))$ *and* $v_2 = (i_2, (t_{2,1}, \ldots, t_{2,n}))$*, the operators are defined, as Colin Fidge proposed, as:*

$$\begin{aligned} v_1 = v_2 &\Leftrightarrow t_{1,i} = t_{2,i} \text{ for } i = 1, \ldots, n \\ v_1 \rightarrow v_2 &\Leftrightarrow t_{1,i_1} \leq t_{2,i_1} \wedge t_{2,i_2} < t_{1,i_2} \\ v_1 || v_2 &\Leftrightarrow \neg(v_1 \rightarrow v_2) \wedge \neg(v_2 \rightarrow v_1). \end{aligned}$$

$=$ *checks for synchroneity,* $\rightarrow$ *for causality and* $||$ *for concurrency.*

For an explanation of how vector time is constructed we refer to the algorithm presented in [CHS97, pp. 20–22]. In the next section we define how vector time is related to the individual events or nodes.

6.1.5 Environments

In section 6.1.6 we will extend the digraph definition 6.1.3 to be equivalent with causality graphs as defined in section 1.2. This extension will be based on the definitions of environments presented in the following.

As mentioned in section 1.2, nodes in causality graphs are decorated with properties (information attached to events) and vector time stamps. In section 5.1 we mention, that the result of evaluating a Wot query is a causality graph with variables bound to nodes. In order to capture this information we define *environments*. Generally, environments are functions which enables us to look up the necessary information about properties, declarations of rules and variables and results from evaluating queries.

Properties and vector time stamps are stored in the *property environment* and *vector time environment*.

Definition 6.1.5 (Property environment) *A property is a member of* **Name***, which is the set of strings consisting of alphanumeric letters. A property environment is a function in the domain* $\mathbf{Env_p} = \mathbf{Node} \hookrightarrow (\mathbf{Name} \hookrightarrow \mathbb{Z})$ *mapping nodes and then properties to a value.*

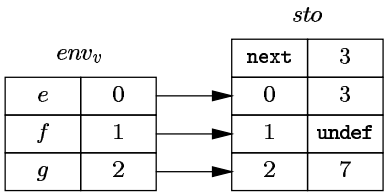


Figure 6.1: The variable environment maps variables to locations in a store. A store maps locations to nodes.

The usage of the property environment can be exemplified with: Given a property environment $env_p \in \mathbf{Env_p}$, we can look up the value of property `tid` on node 5 by evaluating $(env_p\ 5)$ `tid`.

The vector time attached to events can be looked up using the vector time environment which maps nodes to vector time:

Definition 6.1.6 (Vector time environment) *A vector time environment is a function in the domain* $\mathbf{Env_{vt}} = \mathbf{Node} \hookrightarrow \mathbf{VT}$ *mapping a node to its vector time.*

Nodes are bound to variables by the use of *variable environments* and *stores*. A store is a formalization of a memory mapping locations to values and a variable environment is a name space mapping variable names to locations. In order to look up the value of a variable, it is done in two steps by first looking in the variable environment and then in the store.

Definition 6.1.7 (Store) *A location in a store is a member of* $\mathbf{Loc} = \mathbb{Z}$. *A store is a function in the domain* $\mathbf{Sto} = \mathbf{Loc} \cup \{\mathit{next}\} \hookrightarrow \mathbf{Node} \cup \{\mathit{undef}\} \cup \mathbf{Loc}$ *mapping a location to a value. The* `next` *token maps to the next free location in the store and the* `undef` *token denotes that a location is used but in an undefined state.*

As seen in the definition, two special tokens are used to facilitate "memory" management and undefined values. The `next` tokens is always bound to the next free location, that enable us to easily allocate new locations. The `undef` token enables us to see whether or not a variable is in an undefined state.

Definition 6.1.8 (Variable environment) *A variable is a member of* **Name**. *A variable environment is a function in the domain* $\mathbf{Env_v} = \mathbf{Name} \hookrightarrow \mathbf{Loc}$ *mapping a variable to its location in a store.*

The use of variable environments and stores is shown in figure 6.1. This figure shows a situation in which e is bound to node 3, f is in an undefined state and g is bound to node 7.

As seen in the syntax, a query consists of rule declarations, variable declarations and an expression. Variables are maintained through the use of variable environments and stores, as seen above. Rules are maintained in rule environments.

Definition 6.1.9 (Rule environment) *A rule is a member of* **Name**. *An argument list is a member of* $\mathbf{Args} = 2^{\mathbb{Z} \times \mathbf{Name}}$. *A variable list is a member of* $\mathbf{Vars} = 2^{\mathbf{Name}}$. *A rule body is a 3-tuple in* $\mathbf{E} \times \mathbf{Args} \times \mathbf{Vars}$. *A rule environment is a function in the domain* $\mathbf{Env_r} = \mathbf{Name} \hookrightarrow 2^{\mathbf{E} \times \mathbf{Args} \times \mathbf{Vars}}$ *mapping rules to their rule bodies.*

This definition reflects the fact that a single rule name can map to several rule declarations; a rule can have several rule bodies and the rule environment maps the rule to a set of rule bodies, instead of a single rule body if more than one rule declaration is present.

A rule body contains all information about a single rule declaration: the formal parameters, the local variables and the expression to evaluate when instantiating the rule. Formal parameters are stored in a set of 2-tuples indicating the names and positions of the parameters. Local variables are simply stored in a set of names.

6.1.6 Causality graphs

Having the definitions of digraphs and environments, we are now able to define a causality graph, which is a digraph decorated with variable bindings. Using a variable environment and a store, we define it as follows.

Definition 6.1.10 (Causality graph) *A causality graph is an acyclic digraph decorated with a variable environment and a store. It is a 4-tuple* (N, E, env_v, sto), *where* N *and* E *are defined as for a digraph,* $env_v \in \mathbf{Env_v}$ *and* $sto \in \mathbf{Sto}$. *The set of all causality graphs is denoted* $\mathbf{G_c}$.

Here we have extended the definition of digraph with variable bindings and information attached to nodes. Note also, that we now require all causality graphs to be acyclic digraphs—this is the most important property distinguishing causality graphs and digraphs.

6.1.7 Declarations

We use declarations for storing information about variables and rules. This information is stored by updating the environments described previously. Next we give definitions of how this is done.

A variable declaration updates a variable environment and a store. The declared variable must be present in the variable environment, and a location in the store must be allocated to the variable. This location must contain `undef` in order to ensure, that the variable is in an undefined state. Update is performed by the upd_v function.

Definition 6.1.11 (Update variable environment) *A variable environment and a store are updated with a new variable declaration with the function* $upd_v : \mathbf{D_v} \times \mathbf{Env_v} \times \mathbf{Sto} \hookrightarrow \mathbf{Env_v} \times \mathbf{Sto}$*, which is defined compositionally as:*

$$\begin{aligned} upd_v(\texttt{var}\ x; D_v, env_v, sto) &= upd_v(D_v, env_v[x \mapsto l], sto[l \mapsto \texttt{undef}, \texttt{next} \mapsto (l+1)]) \\ upd_v(\varepsilon, env_v, sto) &= (env_v, sto) \end{aligned}$$

where $l = sto\ \texttt{next}$

In the following we will need a function that can tell us the variables declared in a variable declaration.

Definition 6.1.12 (Bound variables) *The bound variables of a variable declaration* D_v *are defined as* $BV(D_v)$*. The function* $BV : \mathbf{D_v} \to 2^{\mathbf{Name}}$ *is defined to be:*

$$\begin{aligned} BV(\texttt{var}\ x; D_v) &= BV(D_v) \cup \{x\} \\ BV(\varepsilon) &= \emptyset \end{aligned}$$

A rule declaration updates a rule environment. This is done by the upd_r function. Information about formal parameters and their order are stored and will be used for mapping parameters by reference when rules are invoked.

Definition 6.1.13 (Update rule environment) *A rule environment is updated with a new rule declaration with the function* $upd_r : \mathbf{D_r} \times \mathbf{Env_r} \hookrightarrow \mathbf{Env_r}$*, which is defined compositionally as:*

$$\begin{aligned} upd_r(r(x_1, \ldots, x_n) \Leftarrow D_v; E; D_r, env_r) &= upd_r(D_r, env_r[r \mapsto S]) \\ upd_r(\varepsilon, env_r) &= env_r \end{aligned}$$

where

$$\begin{aligned} S &= \begin{cases} \{(E, args, vars)\} & \text{if } env_r\ r \text{ is undefined} \\ (env_r\ r) \cup \{(E, args, vars)\} & \text{otherwise} \end{cases} \\ args &= \{(1, x_1), \ldots, (n, x_n)\} \\ vars &= BV(D_v) \end{aligned}$$

As an example, the rule definitions

$$\begin{aligned} p(e,g) &\Leftarrow \texttt{var}\ f;\ e \to f \wedge p(f,g); \\ p(e,f) &\Leftarrow e \to f; \end{aligned}$$

defines two rules named $p(a,b)$ which together describes a path from a to b. These rules have the rule bodies

E	$args$	$vars$
$e \to f \wedge p(f,g)$	$\{(1,e),(2,g)\}$	$\{f\}$
$e \to f$	$\{(1,e),(2,f)\}$	$\emptyset$

The set of these two rule bodies will be bound to the name p in the rule environment. When $p(a, b)$ is used in a query this set will be used to determine the rule bodies to consider in the instantiation process.

6.1.8 Arithmetic expressions

We define the semantics of arithmetic expressions with a denotational semantics. Arithmetic expressions consists of numerals, property lookups and compositions (such as addition, subtraction, multiplication and division). The semantics of numerals are defined as follows:

Definition 6.1.14 *The semantics of numerals are given by the semantic function* $\mathcal{N} : \mathbf{Num} \to \mathbb{Z}$. *Given a numeral* n, $\mathcal{N}[\![n]\!]$ *yields the integer value of the numeral.*

We are now ready to give a compositional definition of the semantics of arithmetic expressions:

Definition 6.1.15 *The semantics of arithmetic expressions are given by the semantic function* $\mathcal{A} : \mathbf{A} \times \mathbf{Env_v} \times \mathbf{Sto} \times \mathbf{Env_p} \to \mathbb{Z}$. *Given* $sto \in \mathbf{Sto}$, $env_v \in \mathbf{Env_v}$ *and* $env_p \in \mathbf{Env_p}$, $\mathcal{A}$ *is defined as:*

$$
\begin{aligned}
\mathcal{A}[\![n, env_v, sto, env_p]\!] &= \mathcal{N}[\![n]\!] \\
\mathcal{A}[\![e.p, env_v, sto, env_p]\!] &= (env_p\ (sto \circ env_v\ e))\ p \\
\mathcal{A}[\![A_1 + A_2, env_v, sto, env_p]\!] &= \mathcal{A}[\![A_1, env_v, sto, env_p]\!] + \mathcal{A}[\![A_2, env_v, sto, env_p]\!] \\
\mathcal{A}[\![A_1 - A_2, env_v, sto, env_p]\!] &= \mathcal{A}[\![A_1, env_v, sto, env_p]\!] - \mathcal{A}[\![A_2, env_v, sto, env_p]\!] \\
\mathcal{A}[\![A_1 * A_2, env_v, sto, env_p]\!] &= \mathcal{A}[\![A_1, env_v, sto, env_p]\!] * \mathcal{A}[\![A_2, env_v, sto, env_p]\!] \\
\mathcal{A}[\![A_1 / A_2, env_v, sto, env_p]\!] &= \mathcal{A}[\![A_1, env_v, sto, env_p]\!] / \mathcal{A}[\![A_2, env_v, sto, env_p]\!]
\end{aligned}
$$

Later we will have use for the function V which gives the set of variables used in an arithmetic expression.

Definition 6.1.16 (Variables in arithmetic expression) *The function* $V : \mathbf{A} \to 2^{\mathbf{Name}}$ *is defined compositionally as:*

$$
\begin{aligned}
V(n) &= \emptyset \\
V(e.p) &= \{e\} \\
V(A_1 + A_2) &= V(A_1) \cup V(A_2) \\
V(A_1 - A_2) &= V(A_1) \cup V(A_2) \\
V(A_1 * A_2) &= V(A_1) \cup V(A_2) \\
V(A_1 / A_2) &= V(A_1) \cup V(A_2)
\end{aligned}
$$

6.1.9 Transition systems

In natural semantics changes in states are the result of evaluating a statement. Such state changes are described by a transition system like the following:

$$< S, s > \rightarrow s'$$

This should be read: *Given state s, the statement S is evaluated resulting in a new state s'.* A statement changes the state to a new state.

Wot works on causality graphs. We define a state in Wot to be a set of causality graphs. A state is the set of all causality graphs having the properties specified by the query evaluated until the present expression. (Remember queries are evaluated from left to right). Based on this, we can now introduce the transition system describing the natural semantics of Wot:

Definition 6.1.17 (Multiple graph transition system) *The multiple graph transition system $\Rightarrow$ has the signature:*

$$g, env_{vt}, env_p, env_r \vdash \langle S, H \rangle \Rightarrow H',$$

where $g \in \mathbf{G_c}$, $env_{vt} \in \mathbf{Env_{vt}}$, $env_p \in \mathbf{Env_p}$, $env_r \in \mathbf{Env_r}$, $S \in \mathbf{Q} \cup \mathbf{E}$, $H \in 2^{\mathbf{G_c}}$ and $H' \in 2^{\mathbf{G_c}}$.

Given a causality graph g with corresponding vector time and property environments, a rule environment, we can evaluate a query from one state H resulting in a new state H'. The transition system is named the *multiple graph transition system*, since a state is a set of causality graphs.

In order to simplify the definition of the multiple graph transition system, we introduce the single graph transition system:

Definition 6.1.18 (Single graph transition system) *The single graph transition system $\rightarrow$ has the signature:*

$$g, env_{vt}, env_p \vdash \langle S, h \rangle \rightarrow H,$$

where $g \in \mathbf{G_c}$, $env_{vt} \in \mathbf{Env_{vt}}$, $env_p \in \mathbf{Env_p}$, $S \in \mathbf{E_s}$, $h \in \mathbf{G_c}$ and $H \in 2^{\mathbf{G_c}}$.

This system resembles the multiple graph transition system, except that in this system, we can evaluate only simple expressions and we distinguish between two kinds of states: a source state (a single causality graph) and a destination state (a set of causality graphs). An expression is always evaluated from a source state and the result of the evaluation is always a destination state.

In the following we will first define the single graph transition system and then use this in the definition of the multiple graph transition system.

6.2 The single graph transition system

In this section we will describe the semantics of the basic expressions in Wot using the single graph transition system. All queries in Wot are built from simple expressions through the use of conjunction, rule definition and rule instantiation.

The first axiom in the transition system is the axiom describing the $\rightarrow$ operator:

Axiom 6.2.1 (Edge)

$$g, env_{vt}, env_p \vdash \langle e \rightarrow f, (N, E, env_v, sto) \rangle \rightarrow H$$

where

$$\begin{aligned} H \;=\; & \{(N \cup \{n_1, n_2\}, E \cup \{(n_1, n_2)\}, env_v, sto') \\ & \mid (\forall i \in \{1,2\} : \textit{If } sto\ l_i \neq \texttt{undef} \textit{ then } n_i = sto\ l_i) \wedge (n_1, n_2) \in E(g)\} \end{aligned}$$

and

$$\begin{aligned} v_1 &= e \\ v_2 &= f \\ l_i &= env_v\ v_i \\ sto'\ l &= \begin{cases} n_i & \textit{if } sto\ l_i = \texttt{undef} \textit{ and } l = l_i, \\ sto\ l & \textit{otherwise.} \end{cases} \\ i &= 1,2 \end{aligned}$$

The evaluation of the edge operator extends the current state with bindings of variables, adds extra nodes and an edge if possible. If either of the variables (e or f) are already bound these bindings are kept. This can be summarized by the following. The destination state H consists of all causality graphs having the following properties:

1. The graph h extended with nodes n_1 and n_2 and edge (n_1, n_2) if needed.

2. The graph is a subgraph of g.

3. e and f are bound to nodes n_1 and n_2 in the graph h if not previously bound.

Note that one graph is added to the destination state H for every possible set of bindings. If both variables are not bound after evaluation an empty state is returned. This reflects the fact that queries are evaluated from left to right and only if all variables in a query can be bound a result is returned. If an expression cannot bind variables the evaluation is terminated.

The axiom for the $\leftrightarrow$ operator is almost identical to the axiom for the $\rightarrow$ operator:

Axiom 6.2.2 (Synchronous)

$$g, env_{vt}, env_p \vdash \langle e \leftrightarrow f, (N, E, env_v, sto) \rangle \rightarrow H$$

where

$$\begin{aligned} H &= \{(N \cup \{n_1, n_2\}, E \cup \{(n_1, n_2)\}, env_v, sto') \\ &\quad \mid (\forall i \in \{1,2\} : \textit{If } sto\ l_i \neq \texttt{undef} \textit{ then } n_i = sto\ l_i) \wedge \\ &\quad n_1 \in N(g) \wedge n_2 \in N(g) \wedge\ env_{vt}\ n_1 \leftrightarrow env_{vt}\ n_2\} \end{aligned}$$

and

$$\begin{aligned} v_1 &= e \\ v_2 &= f \\ l_i &= env_v\ v_i \\ sto'\ l &= \begin{cases} n_i & \textit{if } sto\ l_i = \texttt{undef} \textit{ and } l = l_i, \\ sto\ l & \textit{otherwise.} \end{cases} \\ i &= 1,2 \end{aligned}$$

Here we have added a requirement for the two nodes n_1 and n_2 to be synchronous. This information is looked up in the vector time environment. Since there are no actual edges in the causality graph representing a synchronous event, we only add extra nodes to the new state H.

The $\|$ operator needs no further explanation:

Axiom 6.2.3 (Concurrent)

$$g, env_{vt}, env_p \vdash \langle e \| f, (N, E, env_v, sto) \rangle \rightarrow H$$

where

$$\begin{aligned} H &= \{(N \cup \{n_1, n_2\}, E \cup \{(n_1, n_2)\}, env_v, sto') \\ &\quad \mid (\forall i \in \{1,2\} : \textit{If } sto\ l_i \neq \texttt{undef} \textit{ then } n_i = sto\ l_i) \wedge \\ &\quad n_1 \in N(g) \wedge n_2 \in N(g) \wedge\ env_{vt}\ n_1 \| env_{vt}\ n_2\} \end{aligned}$$

and

$$\begin{aligned} v_1 &= e \\ v_2 &= f \\ l_i &= env_v\ v_i \\ sto'\ l &= \begin{cases} n_i & \textit{if } sto\ l_i = \texttt{undef} \textit{ and } l = l_i, \\ sto\ l & \textit{otherwise.} \end{cases} \\ i &= 1,2 \end{aligned}$$

Here we require the two nodes be concurrent—again by looking at the vector time.

The comparison operators do not differ not much from the above:

Axiom 6.2.4 (Comparison)

$$g, env_{vt}, env_p \vdash \langle A_1\ \theta\ A_2, (N, E, env_v, sto)\rangle \rightarrow H$$

where $\theta \in \{<, \leq, =, \neq, \geq, >\}$ *and*

$$\begin{aligned} H \;=\; & \{(N \cup \{n_1, n_2\}, E \cup \{(n_1, n_2)\}, env_v, sto') \\ & \mid (\forall i \in \{1, \ldots, n\} :\ \textit{If } sto\ l_i \neq \textbf{\textit{undef}}\ \textit{then } n_i = sto\ l_i) \wedge \\ & (\forall i \in \{1, \ldots, n\} :\ n_i \in N(g)) \wedge \\ & \mathcal{A}[\![A_1, env_v, sto', env_p]\!]\ \theta\ \mathcal{A}[\![A_2, env_v, sto', env_p]\!]\} \end{aligned}$$

and

$$\begin{aligned} V &= V(A_1) \cup V(A_2) = \{v_1, \ldots, v_n\} \\ l_i &= env_v\ v_i \\ sto'\ l &= \begin{cases} n_i & \textit{if } sto\ l_i = \textbf{\textit{undef}}\ \textit{and } l = l_i, \\ sto\ l & \textit{otherwise.} \end{cases} \\ i &= 1, \ldots, n \end{aligned}$$

We require that the specified comparison holds between the two arithmetic expressions. This is checked by looking up the semantic value of the arithmetic expressions and comparing these with the θ operator.

6.3 The multiple graph transition system

Having described the semantics of the simple expressions in Wot we are now able to describe the rest of the language. The multiple graph transition systems describes the full semantics of the language.

First of all we use the single graph transition system to define the semantics of all simple expressions in the multiple graph transition system:

Rule 6.3.1 (Generalization)

$$\frac{g, env_{vt}, env_p \vdash \langle E, h_1\rangle \rightarrow H_1 \quad \ldots \quad g, env_{vt}, env_p \vdash \langle E, h_n\rangle \rightarrow H_n}{g, env_{vt}, env_p, env_r \vdash \langle E, H\rangle \Rightarrow H'}$$

where $H = \{h_1, \ldots, h_n\}$, $H' = \bigcup_{i=1}^{n} H_i$, $E \in \mathbf{E_s}$ *and* $H \neq \emptyset$.

A simple expression can be evaluated from a state H in the multiple graph transition system by evaluating it in the single graph transition system from each state (causality graph) in H. The results are then combined by union into H'.

The only axiom in the multiple graph transition system is the stop axiom:

Axiom 6.3.1 (Stop)

$$g, env_{vt}, env_p, env_r \vdash \langle Q, \emptyset \rangle \Rightarrow \emptyset$$

This axiom defines that a query evaluated in the state $\emptyset$ (the empty set) always ends in the same state, meaning that once an expression has failed to make the necessary bindings, the query terminates in the sense that subsequent expressions in the query cannot make new bindings.

The conjunction operator is the glue between subexpressions. It takes results from one subexpression and combines it with the results of another:

Rule 6.3.2 (Conjunction)

$$\frac{g, env_{vt}, env_p, env_r \vdash \langle E_1, H \rangle \Rightarrow H' \quad g, env_{vt}, env_p, env_r \vdash \langle E_2, H' \rangle \Rightarrow H''}{g, env_{vt}, env_p, env_r \vdash \langle E_1 \wedge E_2, H \rangle \Rightarrow H''}$$

where $H \neq \emptyset$.

First E_1 is evaluated from state H resulting in state H'. Then E_2 is evaluated from that state resulting in the final state H''. This illustrates that a query is evaluated from left to right.

To clarify the functionality of the multiple graph transition system a simple example will help. Given an arbitrary causality graph and a simple query $e \rightarrow f \wedge f \rightarrow g$. This query is evaluated from left by first evaluating $e \rightarrow f$. If this evaluation results in two possible bindings of $e \rightarrow f$ represented by the graphs h_1 and h_2 then $H = h_1 \cup h_2$. As the next query is evaluated this is done from the state H. This means that the expression $f \rightarrow g$ is evaluated twice with h_1 and h_2 as two separate source states. If these two evaluations both succeed giving results H_1 and H_2 then the final result of evaluating $e \rightarrow f \wedge f \rightarrow g$ is $H' = H_1 \cup H_2$.

Rule 6.3.3 (Declaration)

$$\frac{g, env_{vt}, env_p, env_r' \vdash \langle E, H' \rangle \Rightarrow H''}{g, env_{vt}, env_p, env_r \vdash \langle D_r; D_v; E, H \rangle \Rightarrow H''}$$

where

$$\begin{aligned} env_r' &= udp_r(D_r, env_r) \\ H &= \bigcup_{i=1}^{n} (N_i, E_i, env_{v,i}, sto_i) \\ H' &= \bigcup_{i=1}^{n} (N_i, E_i, env_{v,i}', sto_i') \\ (env_{v,i}', sto_i') &= upd_v(D_v, env_{v,i}, sto_i) \text{ for } i = 1, \ldots, n \end{aligned}$$

The declaration rule defines the how a query is evaluated by first updating the necessary environments and then evaluating the expression E. As the declaration rule shows, Wot queries with declarations can be evaluated with previous bindings as starting point.

Rule 6.3.4 (Instantiation)

$$\frac{g, env_{vt}, env_p, env_r \vdash \langle E_1, h_1' \rangle \Rightarrow H_1 \quad \ldots \quad g, env_{vt}, env_p, env_r \vdash \langle E_m, h_m' \rangle \Rightarrow H_m}{g, env_{vt}, env_p, env_r \vdash \langle r(x_1, \ldots, x_n), h \rangle \Rightarrow \bigcup_{i=1}^{m} H_i}$$

where

$$\begin{aligned}
h &= (N, E, env_v, sto) \\
h_i' &= (N, E, env_{v,i}, sto_i) \\
s &= env_r\ r = \{(E, args_1, vars_1), \ldots, (E_m, args_m, vars_m)\} \\
args_i &= \{(1, y_{i,1}), \ldots, (n, y_{i,n})\} \\
vars_i &= \{v_{i,1}, \ldots, v_{i,k_i}\} \\
env_{v,i} &= \mathbf{Undef}[y_{i,1} \mapsto env_v\ x_1, \ldots, y_{i,n} \mapsto env_v\ x_n, \\
&\qquad v_{i,1} \mapsto l_{i,1}, \ldots, v_{i,k_i} \mapsto l_{i,k_i}] \\
sto_i &= sto[l_{i,1} \mapsto \textbf{\textit{undef}}, \ldots, l_{i,k_i} \mapsto \textbf{\textit{undef}}, \textbf{\textit{next}} \mapsto ((sto\ \textbf{\textit{next}}) + j)] \\
l_{i,j} &= (sto\ \textbf{\textit{next}}) + j_i - 1 \\
i &= 1, \ldots, m \\
j_i &= 1, \ldots, k_i
\end{aligned}$$

The last definition defines the rule instantiation. This is complex and requires some explanation. First of all it is important to remember that a single rule name can map to several rule bodies. Each of these rule bodies are to be evaluated with an updated (h_i') version of the current state (h) as source state. Each h_i' is updated with separate variable environments. Arguments are parsed by reference. This is handled by letting the new variable environments map the variables of the formal arguments to the store locations of the actual parameters.

When the expressions of a rule body has been evaluated, the variable environments are discarded, but any bindings of actual parameters are kept as these bindings are updated by reference.

6.4 Conclusion

This chapter has presented the formal semantics of the Wot query language. A formal description of the language design presented in chapter 5 has been given. The semantics has precisely stated the functionality of Wot and in addition given a foundation for the processing of queries. In the next chapter the semantics of Wot will be used for constructing algorithms fore evaluating queries.

The semantic description presented was based on the elimination of disjunctions in Wot. We have argued for this simplification of the language and described how Wot models should be transformed only to contain queries built from conjunctions of expressions.

Chapter 7

Prototype

The hypothesis in chapter 3 proposed the use of a language for behavioral abstraction. In chapter 5 we designed the Wot query language and presented the basic functionality through a number of examples and in chapter 6 we gave a formal description of the language which explicitly describes the semantics of Wot.

Having stated the precise functionality of the Wot language forms the necessary basis for the prototype implementation of Wot. This chapter presents the design, implementation and evaluation of this prototype.

With the hypothesis stated in chapter 3 in mind, we state the purpose of the prototype as follows: We wish to test of the capabilities of Wot for abstract modelling of behavior using causality graphs representing actual program executions. Having argued for efficient evaluation of Wot queries we also wish to test the possibilities for an efficient implementation of the processing of Wot queries.

Based on this we formulate the following more specific goals for the prototype implementation of the designed query language:

- The design of the prototype is to focus on the efficient processing of queries. The user of the prototype is not to worry about efficiency when constructing abstract models. The prototype has to optimize the processing of queries to be efficient regardless of the query.

- The capabilities for abstract modelling is tested by implementing only a subset of Wot. We chose only to implement a subset of Wot which will be sufficient for evaluating the fundamental principles of the query language.

- The prototype is to include a simple graphical user interface which can be used for making queries and visualizing the causality graphs and results from queries.

The subset of Wot which we implement is named s-Wot (for small Wot). Compared to Wot this language has all simple expressions but no rules and arithmetic expressions. The possibilities for recursive search in causality graphs s-Wot includes a path ($\twoheadrightarrow$) operator which can be used for modelling paths of arbitrary length. The syntax of s-Wot is defined as:

Definition 7.0.1 (s-Wot syntax)

$$\begin{aligned} Q &= E\,? \\ E &= e\ \theta_1\ f \mid A_1\ \theta_2\ A_2 \mid E_1\ \wedge\ E_2 \mid E_1\ \vee\ E_2 \\ A &= n \mid \text{“}s\text{”} \mid e.p \end{aligned}$$

where $\theta_1 \in \{\rightarrow, \leftrightarrow, ||, \twoheadrightarrow\}$, $\theta_2 \in \{<, \leq, =, \neq, \geq, >\}$. *$n$ is any integer numeral, and s is any string.*

We start by describing the processing of queries. This description deals primarily with the efficient evaluation and is divided into the following sections:

Evaluation of disjunctions Describes how disjunctions are handled.

Data optimization Describes how the physical organization of causality graphs is structured to allow efficient access in queries.

Query optimization Describes how queries are restructured for optimal evaluation.

Search algorithm Describes how the restructured queries are used for searching in causality graphs.

Having introduced the processing of queries we show the actual implementation of the prototype in section 7.5 and give examples of realistic usage of the implemented subset of Wot.

7.1 Evaluation of disjunctions

When a query has been successfully parsed and organized in a parse tree, the query is first transformed into a number of disjunctive queries. This simplification implements the conversion of queries by use of the distributive and commutative laws as described in section 6.1.2.

The algorithm for converting a query represented in a parse tree is described in figure 7.1. This algorithm constructs a list of conjunctions. These conjunctions can be combined in a disjunction equivalent to the original query.

A simple example of this conversion is seen in figure 7.2. This example shows a simple s-Wot query:

$$e \rightarrow f \wedge (f \rightarrow g \vee f \rightarrow h) \tag{7.1}$$

The figure shows the different steps in the conversion. In the algorithm in figure 7.1 we have related these different steps to figure 7.2.

The two trees in figure 7.2d represent the queries:

- $e \rightarrow f \wedge f \rightarrow g$ and
- $e \rightarrow f \wedge f \rightarrow h$.

The final result of the query is achieved by processing each of these queries and combining the result by the union set operation.

The immediate consequence of the conversion is the possibility of getting a number of almost identical queries. From the point of view of evaluation this may seem inefficient when these

1. Push the parse tree of the query on the stack S and make L the empty list.
2. While S is not empty, do:
   (a) Pop the tree T from S.
   (b) Traverse T depth first while doing:
      i. If a conjunction node or a leaf node is encountered, do nothing (see figure 7.2b.)
      ii. If a disjunction node is encountered, make a full copy T' of T. In this copy and in T', remove the disjunction (see figure 7.2c.) Place instead the left subtree of the disjunction in T and the right in T' (see figure 7.2d.) Push T' on S and continue the traversal of T.
   (c) Add T to L.

Figure 7.1: The algorithm for converting a query containing disjunctions to a set of queries with no disjunctions.

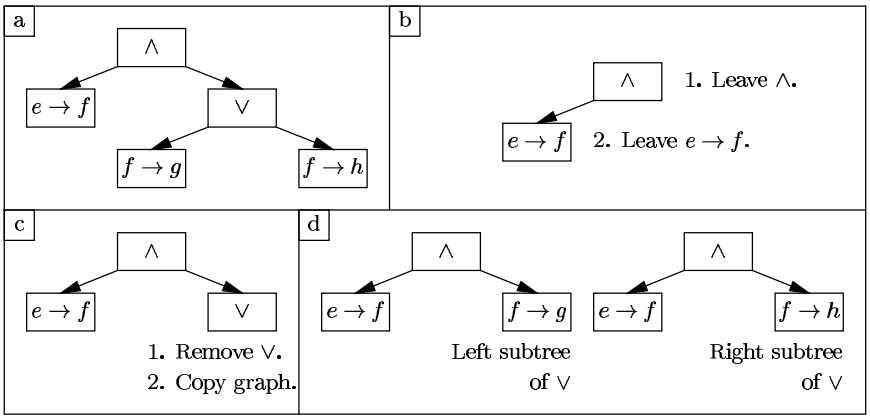


Figure 7.2: Result of converting a the simple query of equation 7.1 including a disjunction.

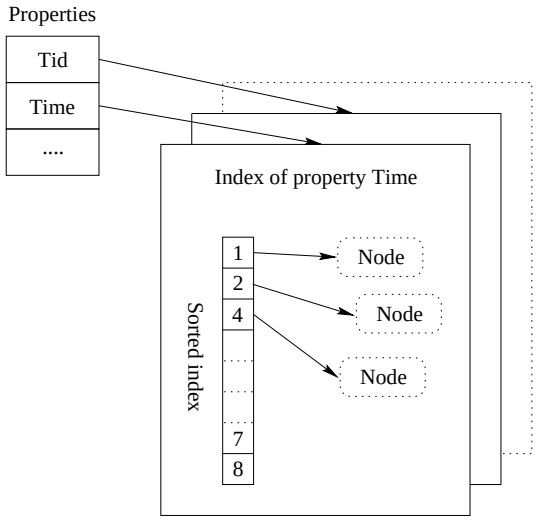


Figure 7.3: Physical organization to allow effecting access to nodes based on properties.

queries are evaluated separately. However the conversion is essential for the algorithms handling optimizations and search. The simplification of queries reduces the amount of bookkeeping needed when queries are evaluated and searching is performed.

7.2 Data optimization

Causality graphs represent a database containing information about application behavior as described in section 5.1.1. To allow s-Wot queries to be processed efficiently we need to structure information about nodes and the attached properties. We begin by giving an overview of the main characteristics of the chosen organisation.

The main structure represent the actual causality graph. The graph is simply structured as a collection of nodes. From each node n there are references to every nodes to which there is an edge from n. There are also references between nodes that are synchronous. This structure ensures efficient access of edges and synchronous nodes. If access to the information in causality graphs was only provided through this presentation, the search for query results would not be efficient. All information about properties and causal relations could only be determined in linear time.

To help this, efficient access to properties is provided through a set of indices, and casual relations can be efficiently determined through the use of time vectors.

7.2.1 Property indices

Nodes in causality graphs are organized according to their properties in indices. To each property there corresponds an index in which all nodes having this property is present. In figure 7.3 this organization is shown.

Each of the indices is sorted according to the values of properties. This allows nodes to be found efficiently given a request for a specific value. The time complexity of these searches is summarized in table 7.2.1. These indices takes up storage in $O(nm)$, where n is the number of

Operator	*First found*	*Next found*
$=$	$O(\log n)$	$O(1)$
$< > \leq \geq$	$O(1)$	$O(1)$
$\neq$	$O(n)$	$O(n)$

Table 7.1: The time complexity of finding nodes in the indices by use of the various comparison operators in s-Wot. The number of nodes in an index is indicated by n. The *first found* column indicates the complexity of finding the first node having a given property, and the *next found* indicates the complexity of finding the successive nodes having the same property.

nodes and m the number of properties. Since m is much smaller than n and in most cases a constant, this representation is quite space efficient.

7.2.2 Vector time

Causal relations between nodes can be efficiently determined by the use of vector time. By having vector time attached to nodes in the causality graph, nodes can be tested for synchrony, concurrency or causality in constant time according to the theory described in section 1.2.

In the prototype we have implemented the post-mortem vector time algorithm which we developed in the TraceInvader project. This algorithm attaches vector time to each node in the causality graph by performing a complete traversal of the graph after it has been created.

7.3 Query optimization

In this section we describe how a query is optimized before it is evaluated through a search. The goal is to ensure that the queries are evaluated as efficient as possible. In order to achieve this, each query is compiled in to an intermediate language which can be evaluated efficiently. During the compilation from a s-Wot query to the intermediate language, the query is optimized in two phases. The first phase is called dependency optimization and the second is called decoration optimization.

In the following we first describe the intermediate language. Then we show the compilation by first describing dependency optimization and then decoration. Section 7.4 describes the use of the compiled query.

7.3.1 Intermediate language

Queries are compiled to an intermediate language. By doing so the optimizations of the query can be expressed, hereby allowing simple but efficient evaluation.

A query in s-Wot describes an abstract model through a number of expressions. Each expression consists of one or more variables. During the evaluation of a query variables are instantiated as nodes with names, and edges are instantiated connecting the nodes.

The expressions of a query can be separated into two categories. Expressions that cause instantiation of variables and edges, and expressions that restrict the values of variables. We call these instantiation expressions and specification expressions respectively. An example of an instantiation expression is the expression $e \rightarrow f$, which causes e and f and an edge to be

instantiated. An example specification expression is $e.time = 5$ which restrict the possible values of the node e.

To get a deeper understanding of some of the problems related to efficient evaluation of queries, we describe a possible evaluation of a non optimized s-Wot query:

$$e \rightarrow f \wedge f \rightarrow g \wedge e.time = 5$$

The evaluation of the query is done by evaluating the expressions from the left to the right. In this case e will be the first variable to be instantiated. But since we have no knowledge of e's value, we select a candidate e node at random. Then f and an edge are instantiated, hereby finishing the first expression. During the evaluation of the next expression g and another edge are instantiated. Finally, the last expression is evaluated causing a of test whether $e.time$ is 5 or not. If the property test fails then we all start over with another initial value of e. It is not hard to recognize the problems of this approach for evaluating queries.

By using the specification expression for locating an instance of e that fulfills the specification, the query could be evaluated more efficiently. This describes the intension of the intermediate language and the purpose of the decoration optimization.

When the query is evaluated, then each of the variables of the query is instantiated exactly once. More precisely each variable is instantiated the first time it occurs in the query when passing from left to right. By binding the appropriate specification expressions to variables at their first occurrence we can use the specification for selecting a value of the variable, and thereby the search space is reduced.

The syntax of the intermediate language express the optimization by allowing specification expressions to be bound to the variables of a query. The following definition gives the syntax for the intermediate language.

Definition 7.3.1 (Intermediate Language Syntax)

$$\begin{aligned}
A &:= B_1 \vee \ldots \vee B_n \\
B &:= C_1 \wedge \ldots \wedge C_n \\
C &:= e\{D\} \mid e\{D_1\} \parallel f\{D_2\} \mid e\{D_1\} \rightarrow f\{D_2\} \mid e\{D_1\} \twoheadrightarrow f\{D_2\} \\
D &:= E_1 \wedge \ldots \wedge E_n \\
E &:= F_1 = F_2 \mid F_1 \neq F_2 \mid F_1 \geq F_2 \mid F_1 \leq F_2 \mid F_1 < F_2 \mid F_1 > F_2 \\
F &:= e.p \mid \textbf{\textit{Value}}
\end{aligned}$$

where A is a disjunction of queries, B is a query, C is an instantiation expression, D is a conjunction of specification expressions, E is a specification expression, and F refers to the values in a comparison.

The primitives are the same as for s-Wot, except a single instantiation primitive. This primitive represent the query consisting of a single specification expression. For example the s-Wot query $e.time = 5$. The solution to this special case has resulted in the C expression $e\{D\}$ (singleton). This expression expresses the instantiation of a single variable, and the compiled query will be: $e\{e.time = 5\}$.

In section 7.3.3 we describe the actual algorithm for decorating the variables with specification expressions.

7.3.2 Dependency optimization

In the following we will describe the dependency optimization. This optimization is an important part of the process of compiling a query to the intermediate language.

In the language, every expression is evaluated from the left to the right. This order is also posed on the instantiations of variables in expressions of the form $e\ \theta\ f$ in which first e is instantiated (if not already instantiated) and then f is instantiated (if not already instantiated). Knowing the evaluation order, helps us to find f efficiently, since we know that e has been instantiated when f is to be instantiated and we have full knowledge of the causality graph. We know that the right variable is strongly dependent of the left variable and this dependency is present because of a θ relation between e and f.

We say that a variable is *dependent* in an expression if we have prior knowledge on the variable in the expression, and we say that it is *independent* in the expression otherwise. Independent variables can only be instantiated by exhaustive search. In atomic expressions of the form $e\ \theta\ f$, f is dependent and e is independent. In general expressions, the dependency of a variable is indicated by its dependency in the leftmost atomic expression in which the variable occurs—this is the expression in which the variable is instantiated.

In the expression

$$g \to h \wedge f \to g \wedge e \twoheadrightarrow f \tag{7.2}$$

the variables e, f and h are independent and g is dependent. If we reorder the expression to

$$e \twoheadrightarrow f \wedge f \to g \wedge g \to h \tag{7.3}$$

only e is independent. Instead of three exhaustive searches only one is necessary thereby reducing the search space significantly. This indicates that it is possible to optimize a query by reordering it in such a way that the number of independent variables is minimized. We will in the following call this optimization strategy for dependency optimization.

The goal for the dependency optimization is to reorder expressions and reduce the number of independent variables. Since any s-Wot expression will be compiled into the intermediate language—which has only instantiation expressions—only subexpressions of the instantiation kind will be reordered. Subexpressions of the specification kind will be left untouched.

In the optimization we use *dependency graphs*:

Definition 7.3.2 (Dependency graph) *Given a conjunction of expressions* $E = \bigwedge_{i=1}^{n} E_i$ *where each* E_i *is on the form* $e\ \theta\ f$, *where* $\theta \in \{\to, \twoheadrightarrow, \|, \leftrightarrow\}$.

The dependency graph for E *is defined as follows:*

1. *The set of nodes is* $\bigcup_{i=1}^{n} E_i$.
2. *There is a directed edge from* E_i *to* E_j *iff the right variable of* E_i *is the left variable of* E_j.

If there is an edge from expression E_1 to expression E_2, then they *share* a variable v—the right-hand-side variable in E_1 which is also the left-hand-side variable in E_2. This means that v is dependent in E_1 and independent in E_2.

The dependency graph is used for reordering the expression. The new order is given by the graph: if there is an edge from E_1 to E_2, then E_1 must be placed before E_2. This ensures

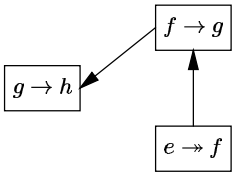


Figure 7.4: The dependency graph for the expression in equation 7.2.

that the shared variable is dependent. If E_2 was placed before E_1 chances are that the variable would be independent instead.

The expression in equation 7.2 has the dependency graph shown in figure 7.4. Using this graph for reordering the expression yields exactly the expression in equation 7.3.

In addition to be used in the dependency optimization, graphs can be used for performing semantic checks of queries. If a graph with variables as nodes and causal dependencies as edges is constructed, this graph can be used for detecting cycles in queries. If a cycle is detected this reflects a query which can never be successfully evaluated, since a causality graphs are defined to be acyclic.

7.3.3 Decoration

In section 7.3.1 we described how specification expressions bound to variables at the point of instantiation could reduce the search space. In this section we will describe the algorithms for placing the specification expressions on the right variables.

During the decoration phase each specification expression is bound to a variable at the variable's point of instantiation. This allows specification expressions to be used for supporting instantiations by specifying the possible values of a variable. Expressions are evaluated from left to right and the basic idea is to use specifications for variables as soon as possible during evaluation hereby reducing the searching needed.

The decoration is performed according to the following simple scheme. For each specification expression a search is performed from the left in the query. When all the variables present in the specification expression has been met then the expression is placed at the last variable seen during the search.

This approach allows variables to be instantiated using the information which can be deduced from the query. The following gives an example of decorating a query that has already been dependency optimized:

$$e \twoheadrightarrow f \wedge f \twoheadrightarrow g \wedge f.tid \neq e.tid \wedge g.tid \neq e.tid$$

Following the mentioned rules for placing specifications on variables, we get the following result:

$$e \twoheadrightarrow f\{e.tid \neq f.tid\} \wedge f \twoheadrightarrow g\{g.tid \neq e.tid\}$$

As explained above the possible values of f are deduced from the specification, because e has been instantiated, when f is to be instantiated. The same holds for g when it is instantiated.

There is a last special case involving the singleton instantiation expressions. If a specification expression include variables that are not used in any instantiation expression, then the specification is transformed to a singleton expression. An example is the query: $e.time = 5$. Which will be compiled to: $e\{e.time = 5\}$.

7.4 Search algorithm

After a query has been compiled it is evaluated through a search. During the search instances of the query are instantiated from the causality graph. This section describes the search algorithm for evaluating the optimized query.

The challenge of constructing the search algorithm, is to find a simplistic and efficient approach for conducting the search. We have tried to achieve these goals by using recursion and backtracking.

The structures of the search algorithm are quite close to the structures used in the description of the formal semantics (see chapter 6.) A causality graph represents the underlying structure for the search. A single result is a causality graph, and full result of a query is a set of causality graphs.

After a query has been optimized it can express a series of searches combined with disjunctions. Each of these searches is treated independently of the other searches. The result of a query is found by combining all the search results from the conjunctions in the disjunction. In the following we therefore focus on evaluating a single conjunction of expressions.

A conjunction of a query compiled to the intermediate language consists of a number of instantiation expressions. These expressions are organized in a parse tree. Finding an instance of the abstract model which the conjunction describes corresponds to performing a successful depth first traversal in the parse tree. A traversal is successful if all expressions in the parse tree has been instantiated. Having decorated expressions with specifications, the expressions are of course only instantiated if the specifications hold.

The details of this algorithm are explained below using a pseudo Lisp notation for functions for list manipulation. Instantiation expressions of a query corresponds to list elements. In the description of the algorithm the following are used: g represents a causality graph, h represents a subgraph of g —corresponding to a single state in the transition systems described in section 6.2—and L represents a query as a list (conjunction) of expressions.

The algorithm is defined by the function $\mathit{eval}(g, h, L)$. This function evaluates the first expression in the list L and calls itself recursively to evaluate the rest of L.

The function $\mathit{eval}(g, h, L)$ is defined as:

1. If L not empty: For each instance of $\mathit{head}(L)$ in g:
   (a) Add the instance to h (if the instance is not already present).
   (b) Call $\mathit{eval}(g, h, \mathit{tail}(L))$.
   (c) Remove instance from h if added in 1.
2. Else: The evaluation has finished. Remember the result h.

When an instance of $\mathit{head}(L)$ is found then $\mathit{eval}(g, h, L)$ is called with a copy of h. If no instances of an expression can be added to h then h is discarded and the algorithm backtracks to continue the recursion. When L is empty then h represents a result of the query L.

When a path expression ($\twoheadrightarrow$) is evaluated a different recursive search is performed before each expression of this type is instantiated. This recursive search performs a breadth first traversal of the causality graph to find instances of paths.

The use of specification expressions makes the search efficient. These are used when searching for an independent variable. An example: when searching for the variable e in the query $e\{e.time = 5 \wedge e.tid \neq 1\} \rightarrow f$. Here nodes with the property $time = 5$ are found using the

indices in the causality graph. Each of these candidates is then tested against the rest of the specification expressions, before it is actually used as an e variable.

To minimize the number of tests on candidate variables, one must find the specification expression that gives the smallest number of results. It is not possible to determine which specification expression gives the smallest result, and therefore operator precedence is used. It is more likely that an equal expression gives a smaller result than a not equal expression. The operator precedence order is the following, starting with the expressions that are likely to give the smallest result: $=$, $<$, $>$, $\leq$, $\geq$, $\neq$.

Complexity

Table 7.2 describes the different time complexities for finding all possible instances of an instantiation expression in a causality graph. n is the number of nodes in the causality graph, k the maximum out-degree of a node, m the maximum number of nodes involved in a synchronization, and l the maximum length of a path.

Expression	Initiation	Completion
$e \rightarrow f$	$O(n)[O(1)]$	$O(k)[O(1)]$
$e \| \| f$		$O(n)[O(1)]$
$e \leftrightarrow f$		$O(m)[O(1)]$
$e \twoheadrightarrow f$		$O(k^l)[O(k^l)]$
e		$-$

Table 7.2: The time complexity of instantiation expressions.

The instantiation column describes the worst case complexity of finding all possible instances of the first variable in an expression. The complexity given in the brackets describes the complexity when the first variable is bound.

The completion column describes the worst case complexity for finding second variable of the expression when the first variable has been instantiated. Again, the complexity given in the brackets describe the complexity when the second variable is bound.

Path expressions are different from the rest of the instantiation expressions. The difference lies in the number of different instances when the end nodes have instantiated. Opposed to the other expressions, the path expression can have more than a single result when the end nodes are instantiated. Therefore the bracketed complexity of completing a path expression is quite large.

7.5 Graphical user interface

The last sections gave an in-depth description of the physical organisation of causality graphs and the processing of queries. In this section we describe the graphical user interface and show the usage of s-Wot in an example.

7.5.1 Design goals

We begin by defining the overall goals of the graphical user interface. As required in the introduction this prototype is to include a simple graphical interface for constructing queries

and displaying results of queries. Since we have chosen to focus on the implementation of the processing of queries only the basic functionality for visualization is to be provided.

The main purpose of the user interface is to:

- provide the user with an interface for constructing s-Wot queries which then can be processed according to the description in the previous sections.
- display the causality graphs of which abstract models are to be constructed. Provide basic functionality for extracting information from causality graphs such as properties attached to nodes.
- display results from queries and provide a relation between results and causality graphs.

The analysis module which we developed in TraceInvader included an advanced user interface with several possibilities for visualizing the behavior of a distributed application. However, it is not our intension to develop a similarly complex user interface in this prototype.

7.5.2 User interface

The s-Wot user interface consists of a single window with a section for typing in queries and a section for visualizing causality graphs and the result of queries. See figure 7.5. Once a causality graph has been loaded the whole graph is automatically displayed and queries are then ready to be typed in.

When the query is constructed and evaluated the result can be used for changing the query to perform a new search or the user can continue to search for the next possible result.

Graphs returned from a query are characterized by possible variable bindings to specific nodes in graphs. In the developed user interface we have chosen to visualize results as a highlighted subgraph in the visualization of the whole graph. Possible variable bindings are attached to nodes in the visualization. This supports the logic behind constructed queries.

Information attached to nodes can also be extracted directly from the causality graph by clicking on nodes on the visualized graph. This functionality was also included in TraceInvader. Having this functionality present is important, even though information is to be extracted from the causality graph mainly through the use of queries. The view of causality graphs and highlighted results can as in the TraceInvader be zoomed in and out.

7.5.3 Example

The bitonic sort algorithm which we will use is an example of a distributed application with many processes and extensive communication between processes. We will use a trace of this algorithm to show an advanced example of abstract modelling of behavior. In the causality graph representing the trace five different properties are attached to events: *time* (local lamport time), *line* (source code relation), *type* (of event), *file* (relation to file) and *tid* (task identity). These properties will be used in the model. For a view on the whole graph see the screen shots provided in the appendix of [CHS97, App. A].

The algorithm uses a set of processes ($tid = 262161 \ldots 262168$) organized in a cube to participate in the sorting of an array of numbers. When the sorting starts each process in the cube is spawned. The master process ($tid = 262159$) then begins to hand out sorting jobs and waits for the results to be returned.

In this example we will construct a model able to check if every slave process has been properly started from the master process and has returned a result to the master.

Bitonic s-Wot-model

The s-Wot model is given as the conjunction of the following expressions which will be explained below:

1. $a.tid = 262159 \ \wedge \ a.type = PvmSend$
2. $a \rightarrow b \ \wedge \ b.tid \geq 262161 \ \wedge \ b.tid \leq 262168 \ \wedge \ b.line = 138$
3. $a \ \rightarrow \ c \ \wedge \ c.tid = a.tid$
4. $c \ \rightarrow \ d \ \wedge \ d.line = 141$
5. $c \ \rightarrow \ e \ \wedge \ e.tid = a.tid$
6. $e \ \rightarrow \ f \ \wedge \ f.line = 148$
7. $f \ \twoheadrightarrow \ g \ \wedge \ g.line = 156 \ \wedge \ g.tid = b.tid$
8. $g \ \rightarrow \ h \ \wedge \ h.tid = a.tid$

We begin by modelling how processes should be properly started. This is done by three successive communications from the master process to any of the slaves. Each of these communications is checked properly by stating conditions for properties.

First the beginning of the algorithm is modelled in expression 1 where event a is required to be in the master process. Then the first message to any slave process is modelled in expression 2. The identity of the first message is modelled by property $line = 138$. Expression 3 models that the next message is subsequently following the first message and in expression 4 the next message is modelled through events c and d. The second message is identified by the property $line = 141$. A similar scheme is used for the last message in expressions 5 and 6.

When a slave process has been properly initialized the actual sorting is performed. In expression 7 we model how processes finishes sorting and reaches a result. During sorting a number of internal and external (send/receives) is to be performed before a result finally is returned to the master process. In this model we are not interested in the details of the communication performed by the sorting process. We are only interested in modelling whether a result is actually returned to the master process. This is modeled by an arbitrary path from event f to event g where g indicates a the achievement of a result (property $g.line = 156$). Finally a result is returned to the master process. This is modelled in expression 8.

When this model is typed and evaluated the query is processed according to the algorithms in the previous sections. The results returned are, as described, highlighted in the visualization and variables are attached to nodes. An actual result from this query can be seen in figures 7.5 and 7.6. In figure 7.5 the initialization of process is seen. The result of this query returns all edges and nodes specified in the model. In figure 7.6 the returning of a result is seen. The screen shots show the visualization of a single result. If the next button is pushed the next possible result is visualized.

This example clarifies two main issues. First of all the example shows how s-Wot has enough power of expression to allow advanced models to be constructed and illustrates how the simple constructs can be used to construct models which captures complex behavior.

Secondly the example justifies the need for the abstraction. The path operator only enables paths to be specified according to the start and end nodes. In the example this is indicated by node variables f and g. The path between these nodes cannot be further specified. If the prototype implemented the full Wot language this could be specified using recursively declared rules. As described in chapter 5 this would enable invariants to be placed on queries.

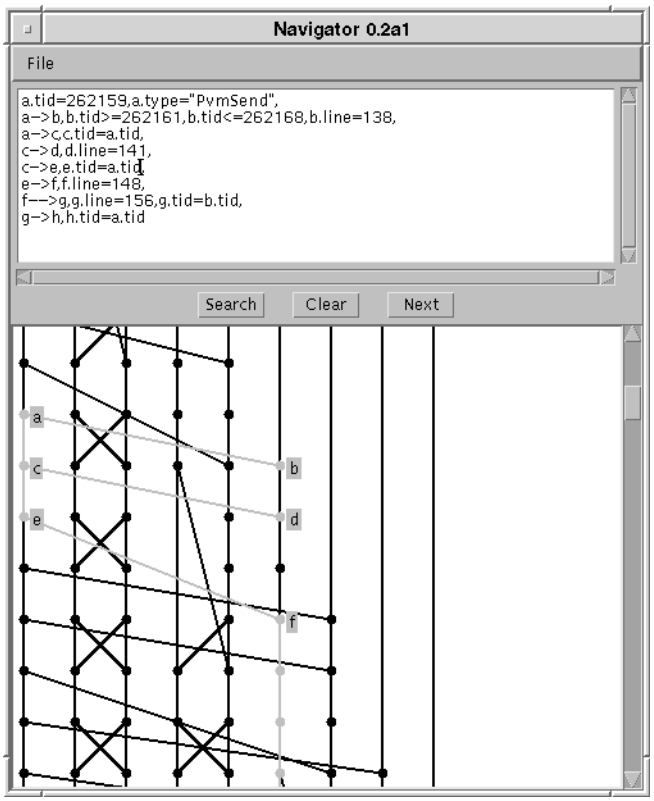


Figure 7.5: An example of behavioral abstraction in a causality graph of an advanced sorting algorithm. The screen shot shows the initialization of slave processes.

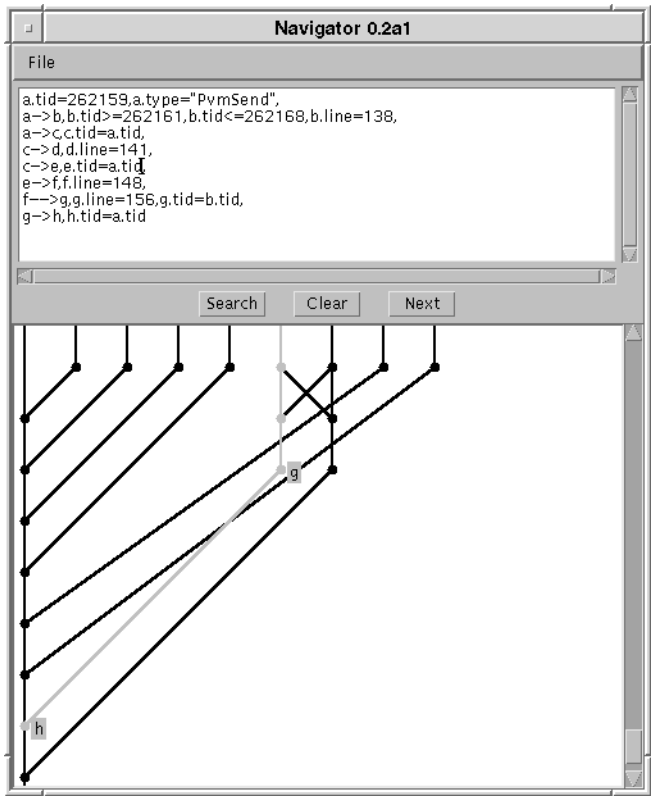


Figure 7.6: The screen shot shows the returning of a result to the master process.

7.6 Conclusion

In this chapter we described the prototype implementation of s-Wot—a subset of Wot including a path operator but with no rule declarations. The main purposes of the prototype has been to test the capabilities of s-Wot for abstract modelling and the possibilities for efficient processing of queries.

The design of the efficient processing of queries has generally proven to be very successful. Given a s-Wot query and a causality graph, the prototype will find all possible results of the query. By having indices on properties the processing is made efficient. The algorithms which we have developed has proven successful. Also the implemented strategies for optimization of queries have generally been successful. However work still remains to be done in this field, even though we believe that the optimization strategies developed here can be a strong foundation for further work.

Through the example in this chapter we have seen a realistic use of abstract modelling of behavior in s-Wot. Even though we have not implemented the complete Wot language the implementation s-Wot has shown how simple expressions are useful for creating models of behavior closely related to the concepts of causality graphs.

Chapter 8

Conclusion

This project was initiated by the problems related to debugging distributed applications. During the last three semesters we have analysed the general problems of debugging distributed applications and studied the proposed theories for solving these problems. Based on this study we have extended these theories and proposed new approaches to debugging. In the TraceInvader project which was the first part of our masters thesis we implemented the theories and developed a debugging tool. The main result of the TraceInvader was the proposal of causality graphs.

In this, the final part of our masters thesis, we have developed further theories for debugging distributed application. This work has been based directly on the theories of causality graphs and the main result of has been the development of powerful language for behavioral abstraction.

The hypothesis which we have tested was stated in chapter 3 as follows:

1. The complexity of extracting information from a causality graph can be reduced through the use of a simple language for describing abstract models of behavior.
2. Such a simple language can be implemented in an efficient way and support the automatic detection of abstract models of behavior in a causality graph.

In the following we summarize the results presented in this thesis and reflect on whether the hypothesis has been successfully proven. After this we present ideas for further work based.

8.1 Causality graphs and behavioral abstraction

This project is based directly upon the TraceInvader project in which we came up with our first contribution to solving the problems of reliable observation and complexity in debugging distributed applications.

Through efficient algorithms for maintaining causal relations and vector time, we handled the problem of reliable observation. Our proposal for handling the problem of complexity was the use of behavioral abstraction.

Several layers of abstraction was posed on the behavior. The different levels of abstraction was the execution level, the trace level, the causality graph level and the user level. The execution level represents the events occurring in an application and the trace level introduces a filter on the execution level. The causality graph represents the causal relations between events at the execution level and the user level uses graphical visualizations of the causality graph for

gaining an overview of the behavior. At the user level we defined functionality for pruning causality graphs in order to reduce the amount of information visualized—this way a user-guided reduction in complexity was possible. The user can prune the graph in order to see if a specific behavioral pattern is present.

In this project we have taken behavioral abstraction one step further. The main problem with the user level as it is designed in the TraceInvader project is that it is quite cumbersome to prune the graph. The pruning is done through functionality for hiding and showing parts of the graph and it is done entirely in the hand. This manual pruning does not reduce the complexity of debugging significantly. The levels of abstraction has to be raised.

In the TraceInvader II project we have focused on developing methods for finding behavioral patterns through the use of language defined behavioral abstractions. In contrast with manual pruning, these can be reused. It has also been shown in this project that it can be automated.

We stated in the hypothesis that a simple language for defining behavioral abstractions can reduce the complexity problem in debugging distributed applications. We also stated that it is possible to develop an efficient implementation of this language. In order to test this hypothesis we have designed the Wot query language and implemented a prototype version of the language. In the following we will describe the results gained from the design of the query language and the prototype implementation.

8.2 The Wot query language

The Wot query language has been designed with focus on fulfilling the hypothesis, by being a simple language for describing abstract models of behavior.

Elaborating further on the hypothesis additional requirements were set for the language. These are mainly focused on the importance of handling abstraction, relation to theory, and general language design issues such as simplicity and regularity. Abstraction is the most important requirement for the Wot query language since this is what gives the language the strength to handle the complexity problem.

The design of Wot has been inspired by the Prolog programming language. The main reason for this is Prologs ability to handle abstraction through the use of rules and compositional definitions.

By using the principles of Prolog the Wot language for describing abstract models of behavior has been developed. The language supports the reasoning possibilities that are provided through the causality graph, hereby allowing advanced comparisons between modes in a causality graph. Furthermore the language is general, because the design focus on causality graphs instead of the task of debugging distributed applications. This allows the language to be used in other applications involving causality graphs, such as version control systems. Regularity has been a natural part of designing the language, and is especially displayed in the uniform treatment of expressions.

The natural semantics for the query language has been given. This provides a precise understanding of the query language. Furthermore the semantics of Wot provide the algorithms for evaluating and searching for instances of abstract models of behavior in causality graphs.

Overall we conclude that the designed language fulfill the requirements concerning abstraction, simplicity, generality, and regularity.

8.3 Prototype

In this report we have presented a prototype of an implementation of the query language Wot. The language which we implemented is s-Wot which is a subset of Wot. The constructs removed from Wot are rules and arithmetic expressions. Only the most basic operators are left—the focus in the prototype was on evaluating only the most basic expressions. We added a path operator for finding paths in order to have one example of a more complex abstraction and evaluated the performance of such an abstraction.

The primary purpose of the prototype was to examine whether it is possible to develop techniques for processing queries and make an efficient implementation of these techniques.

In this project we have described and implemented query processing which uses the following techniques for gaining efficiency:

- Data optimization: This covers efficient access to nodes in causality graphs by the use of indices and time vectors.
- Query optimization: This covers the compilation to the intermediate language.

The search algorithm presented uses the information from these two optimizations in making an efficient search.

It has been shown that it was possible to make an efficient implementation of the prototype language s-Wot. This partly proves the parts of the hypothesis concerning the implementation of Wot. In order to prove the hypothesis fully, it is necessary to make a full implementation of Wot and evaluate this.

8.4 Summary

The overall result of this project has been the development of a language for behavioral abstraction in causality graphs. This language has been proven to be usable and efficiently implementable. The language is capable of handling the problem of complexity in debugging distributed applications—if used correctly.

Combined with a user-friendly graphical user interface we believe that this language, its implementation and the work we have done in the TraceInvader project is a strong foundation for a debugging tool for debugging distributed applications.

8.5 Future work

The future work of the current results can be divided into three categories: Extending the language with additional functionality, completion of the algorithms and using the language in a debugger.

Extending the language

Generally it is our opinion that the current language is sophisticated enough for describing advanced models, but we have an issue that could be useful for describing more advanced models.

The current query language does not allow any other types of variables than variables of type node. By allowing for example integer variables, one could use counters for describing abstract

models. For example one could say that exactly five instances of a model should occur before another model occurs.

More complex ways of putting constraints to models could also be powerful. This could for example be the use of universal quantifiers ($\forall$) and existential quantifiers ($\exists$).

Completing the algorithms

The optimization algorithms that we have provided do not cover a language with procedure declarations. Before implementing the full language these algorithms need to be designed for automatically achieving efficient evaluation of queries using procedures.

Using the language in a debugger

The graphical user interface in the prototype is mainly intended for testing the expressibility of the designed language. It does not take advantage of the abstraction mechanisms provided through the language. It would be interesting to design a debugger that takes advantage of the possibilities of the language. It is our opinion that such a debugger could be a strong tool for handling the complexity problem, because of the support for abstraction and automation.

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.

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

[CHS97] Mikkel Christiansen, Jesper Hagen, and Kristian Qvistgaard Skov. TraceInvader—a tool for debugging distributed applications. Master's thesis, Aalborg University, Department of Computer Science, Fredrik Bajers Vej 7E, DK-9220 Aalborg Øst, Denmark, January 1997.

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

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

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications—A Formal Introduction.* John Wiley & Sons, 1992.

[SS86] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* The MIT press, 1986.

[vD80] D. van Dalen. *Logic and Structure.* Springer-Verlag Berlin Heidelberg, 1980.