Development of an Analysis tool for execution traces

Masters' Thesis

Anders Johnsson
Mälardalens Högskola
Västerås, Sweden
ajn00012@student.mdh.se

Roy Nilsson
Mälardalens Högskola
Västerås, Sweden
rnn00002@student.mdh.se

6 September 2004

Abstract

In this thesis we present the *Probabilistic Property Language* (PPL) and a supporting tool. The purpose of this language is to analyse real-time systems based on information in execution traces. Traditional real-time analysis, e.g. fixed priority analysis, tends to be too pessimistic. Systems that does work can sometimes be deemed unschedulable since only worst case times, e.g. execution times or minimum inter arrival times, are considered. This is a problem when the worst cases are unlikely or never occur in the system. Furthermore these methods of analysis have no mean of checking properties like non emptiness of a message queue. To deal with that a simulation based analysis method was developed [13][14]. Using a modelling language a model of the system is constructed. Changes can then be made in this model before they are implemented in the system. Executing this model in a simulator will result in a trace. The trace contains information about task switches and probe observations that will be used to analyse the system with the help of PPL. A PPL query is formulated as a probabilistic statement, stating for example that some property should always be true, e.g. meeting a hard deadline, would be to formulate it as having a probability of 1. The probe observations contain information about system resource usage and other more application specific information. They could for example contain the size of a message queue. This could be used to verify requirements that could not be checked using fixed priority analysis like for example a message queue never being empty. Many different properties like precedence, separation and pre-emption can be formulated using PPL. In addition to this the query can contain an unbounded variable. This variable can be used to retrieve various constraint values, e.g. to find what deadline is met with a given probability.

Contents

1 Introduction

In order to analyse temporal properties in a large and complex real-time system, one method is to gather execution traces from the system and base the analysis on the information in those traces. These execution traces would contain very large amounts of information making them practically impossible to analyse manually. Therefore a query language, the Probabilistic Property Language (PPL), has been outlined to assist in this analysis.

1.1 Background

Many large and complex real-time systems have evolved over a long period of time. Over time functionality has been changed and new features added. Eventually the temporal model of the system, if there were one in the first place, will no longer be consistent with the current system. In the early stages this might not be a problem as the system is still rather simple and the effects of making changes can quite easily be analysed. However as the system grows analysing becomes increasingly difficult to the point where the system will need to be re-engineered to reintroduce analysability.

An attempt to do this in a robot control system at ABB Robotics was done by Wall, Andersson and Norström [13][14]. At first traditional real-time analysis, FPA (fixed priority analysis), was considered. However these methods of analysis were found to be insufficient. FPA give a true or false answer to the question "is the system schedulable?". In this case the system was deemed unschedulable while in practice it is schedulable, i.e. the FPA was too pessimistic. That is because of the FPA only using worst case times, e.g. worst case execution time or minimum inter arrival time. Furthermore in this system there are other, additional, correctness criterions. One such criterion is a message queue that may never be empty. No analysis method was found that supported such criterions. Hence a different approach was needed.

A simulation based approach was chosen. Using simulation those other criterions can be analysed. In addition the distribution of execution times, rather than the worst cases, could be used in the analysis. Software probes are used to measure the system and these measurements are then written to a log file after the measuring is done. The logged information is: changes in value on general probes, time when task start executing, time when task was interrupted, time when task restart after interruption and time when task finish execution.

A modelling language, ART-ML (Architecture and Real-Time behaviour Modelling Language), was developed. Using ART-ML the system is modelled and when changes are to be made they can first be introduced in the model. Simulating this model of the system will result in a log file similar to one from probing the actual system.

From these log files properties of the system can be extracted. However because of the size of the files it is not feasible to do it manually. Thus a tool was required to perform the analysis. To perform the analysis a probabilistic query language, PPL (Probabilistic Property Language), was outlined. PPL queries are written as probabilistic statements about a property. Checking, for example, that a task t

always meets a deadline of 10 would be written as the response time of t being less than 10 with a probability of 1. Furthermore, PPL support the use of unbounded variables. Unbounded variables can be used to return values to the user.

1.2 Problem description

Our task consists of two parts, to define PPL in detail and to develop a PPL tool.

First part is to define the PPL language. To define all the operators; their syntax and exact semantics. To define the data model; what information is available and how is it accessed. The use of the instance operator. Furthermore how and where unbounded variables may be used as well as how they are bounded.

The second part is to develop a PPL tool. This tool is to determine if a PPL query is true or false with respect to a given execution trace. If the query contained an unbounded variable it will be bounded and the bounded values returned to the user. Possible additional features could be for the tool to support macros in order to make complex queries easier to formulate. Another one could be to, in some manner, simultaneously work on several traces.

1.3 Related work

1.3.1 Log file analysis

Log file analysis has been used in several testing-related projects. Qiao and Zhang [12], use a log file analyser to check communication consistency in a parallel system. Their tool is made specifically for programs using MPI (Message Passing Interface). They have implemented a logger that, during runtime, logs all communication events. In addition they have a wrapper that extends the original MPI functions to support the logger. The wrapper controls the logging process; it chooses what to log in situations. It also starts and stops the logger on certain MPI function calls. The last step is applying the analyser. This is done post execution. Each process in the parallel system generates its own log. The analyser takes these logs as input and check for matching pairs of events. Each found pair is moved to a "complete list". If all events from the logs where moved to the complete list then there where no errors in communication during this system run. Although their tool is made for MPI, Qiao and Zhang claim it could easily be adapted to other message passing libraries.

Andrews and Zhang, [3][4][5], has taken a more general approach to log file analysis. Their approach is to create a log file analyser as set of state machines. Each machine checks some specific requirement of the tested software. The machines will ignore the lines of a log file that is not relevant for their purpose. For each line that a machine recognizes it makes a state transition. An error is found when a machine recognizes a line but has no valid transition for it from its current state. To create analysers they have defined a language, LFAL (Log File Analysis Language). LFAL is used to define states and transitions to create state machines in a straightforward way. In addition they have created a complier to make executables of LFAL analysers. Their work does not concern the logging process. Instead they assume the tested software writes proper records to the log file. The LFAL analysers set few limitations on the log files they are to analyse. Each line in the files is to consist of a sequence of

keywords, strings and/or numbers, beginning with a keyword. Such general requirements allows for a variety of different kinds of log files. Thus LFAL analysers could be applicable in a very wide spectrum of tests. In one of their papers, [4], they present the result of a study where they constructed and applied a LFAL analyser to a complex safety critical system, the steam-boiler control system. The analyser worked well which they believe partly was a result of the compositionality of the analyser. The problem was broken down and spread over several state machines. However they recognize that "more experience is needed to say whether safety properties are usually amenable to this kind of analysis".

In his work on Rapid Application of Lightweight Formal Methods for Consistency Analyses [8], Feather conducted two studies on NASA spacecraft software. This software was divided into larger modules. Each of these modules where designed and developed by different teams. His first study analysed the interfaces the different modules used to communicate with each other. Each modules interface is modelled in a database. Inconsistencies in the interfaces could be found by issuing queries on the database. As an example an inconsistency could be a module M1 having a message MSG listed as outgoing to module M2 when M2 do not have MSG listed as incoming from M1. Because of the large quantities of data to analyse a database approach was a good choice. And since the calculations needed for the analysis where quite simple a database tool was sufficient. As his work was on rapid analysis Feather needed a database tool with a powerful query mechanism. His choice was AP5, a research-quality database tool developed at University of Southern California. According to Feather AP5 is flexible and powerful. The second study was to analyse log files created by the same software during its execution. The log file held records of all messages passed between the different modules. Again the same database approach was used. The log file was loaded into a database. Then by issuing queries on this database, violations of the systems requirement could be found.

1.3.2 Data mining

Bratko and Šuc has done work on machine learning from numerical data [6]. In this work they use an approach involving qualitative data mining to find qualitative patterns or relationships. They reason that building a quantitative model of a complex system is often a demanding or even unrealistic task. The task would be simpler if the problem could be solved at a qualitative level of abstraction. For that they present an approach using a learning program called QUIN (Qualitative Induction). QUIN is used to search for patterns in numerical data. These patterns are then combined into qualitative trees. Using induction of the tree a solution is found. In the mentioned paper they present a case study where a system learned to control a crane by learning from traces of human operators controlling it.

The fields where data mining is applicable is wide. Because of that there exist a variety of data mining tools. Han et al, [9], believes that these tools are only interfaces build on similar underlying mechanisms. They make comparisons to the success of relational databases where a standardized relational query language was developed early on. Hence they have designed a data mining query language, DMQL. The language was designed based on these five considerations. The set of relevant data and the kinds of knowledge to be discovered should be specified in the data mining request. Background information could be available to help in the mining process. The results should be

specified in generalized or multiple-level concepts rather than primitive data. And there should be the possibility to specify various thresholds to filter the results.

Data mining has been used to find sequential patterns in databases. In [1] Agrawal and Srikant present their work on finding patterns in a large database of customer transactions. Their task was to find connections between items; what items were bought in sequence. They use one example where a customer renting the first star wars movie is likely to also rent the second and third, however not necessarily at the same time or even consecutively. For this task they have developed a five phase algorithm. In the first phase the database is sorted on customer and transaction time. In the second phase the transactions are split into item sets. In the third phase each customer's transaction sequence is transformed into a sequence of item sets. In the next phase the item sets are used to find desired sequences. They have three algorithms for this phase, each with their properties making them more or less suitable depending on the properties of the database. In the final phase the maximal of the sequences found in the previous phase is found.

2 The PPL Language

The greater purpose of the *Probabilistic Property Language (PPL)* is to analyse the impact of changes made in a real-time system. The changes are introduced in a model and a simulation of this model results in an execution trace. The need for a language like PPL is because of the size of these traces. To try to manually gather any useful information from them is simply not an option.

Using PPL, properties of tasks and message queues can be extracted. The properties are extracted as probabilities of fulfilling some requirement. To find out if some task t always meets its deadline of 10 time units would be to ask if the probability, of the response time of t being less than 10, is 1.

$$P(t(i),t(i).response < 10) = 1$$

Example 2.1 task t should always meet its deadline 10

This way the analysis of the changes is a process of, from the trace, count and compare the values collected during the simulation. A task t always meeting its deadline of 10, as in Example 2.1, would be done by checking if the response time for all instances of t is less than 10. That is, all observed response times for t is less than 10.

An outline for PPL and its semantics was previously presented, along with a grammar, in [14]. In Appendix A we present our version of the PPL grammar. Naturally the foundation of our grammar is the same as the outlined version. Most noticeable difference is a syntactical change to the probability function, P, to help avoid ambiguity. In addition we have extended it with the statistical functions *min*, *max*, *avg* and *median*. For the instance operator the function *following* has been added to better compare different tasks. The instance operator has also been extended with a macro to simplify properties over sequences . We also allow more complex arithmetic expressions

2.1 Queries

PPL is intended to be used to formulate probabilistic queries. A probabilistic query can be defined as a relational operation on two probabilities.

```
<probability> <relation operator> <probability>
```

Syntax 2.1 query

A probability can be either a constant between 0 and 1, the *probability function P* or an *unbounded variable* (see Section 2.7). The result from a query without an unbounded variable is either *true* or *false*. If the query contains an unbounded variable then the result is the value or interval of values on the variable for which the query is true. In addition to the probabilistic queries a query is allowed to

consist only of one of the *statistical functions*. Finally there is a function called *subset* that writes all values in a set to a file, e.g. it could write all response times of some task.

2.2 Task

In PPL a *task* is considered as a set of *instances*. Each instance being one execution of that task in the system. In order to analyse a trace the information about its task instances need to be accessed. Each instance has four basic data members, *start time*, *end time*, *response time* and *execution time*. The trace can also contain *probes*. These probes are general and the value from each probe might have different meaning for different traces. It could for example be the number of messages in a queue. It is up to the user to know what information was observed by the probes. The probes can be accessed as data members to get the value of that probe when the task instance begun execution.

Data members	
start	The time when this instance of the task begun executing.
end	The time when this instance of the task was done executing.
response (resp)	The response time of this instance. The time passed from starting execution to finishing execution.
exec	The execution time of this instance.
probe[16..255]	The value the probe had when this instance begun execution. The id of a probe can range from 16 to 255.

Figure 2.1 data members

The syntax for tasks, in PPL, is shown in Syntax 2.2. The name of the task is used to specify what task. With the *instance operator* an instance of the task is selected. To access data of an instance a "." and the name of the data member is used, like when accessing a field of a C-struct.

```
<task>(<instance variable>).<data member>
```

Syntax 2.2 accessing data members

```
t(i).resp
```

Example 2.2 accessing the response times of task t

2.3 Instance operator

As mentioned in Section 2.2, a task is a set of instances where each instance is an execution of that task. When evaluating PPL queries it is these task instances that are compared. The probability of a task fulfilling a temporal requirement is how many of the instances that fulfil this requirement. The instance operator is used to bind instances. Since all instances need to be evaluated to calculate a

probability it is not possible to explicitly state what instance to compare with, i.e. writing “t(1).resp” is not valid. Instead a variable is used in the instance operator in order to compare instances. To compare relative instances an integer may be added or subtracted to that variable.

In Example 2.3 the response time of all instances is compared to the response time of the next instance by adding 1 to the instance variable.

```
t(i).resp > t(i+1).resp
```

Example 2.3 comparing instances

The instance operator could also be used to compare the same instance of different tasks as shown in Example 2.4.

```
t(i).resp > s(i).resp
```

Example 2.4 comparing tasks

2.4 Probability function P

The function P is the core of PPL. It is the foundation of all probabilistic queries as one without P would only compare constants. P takes two arguments, first the working set, the task from whose point of view the function is formulated, the second argument is the condition of the function. The result from P is the probability of an instance of the set fulfilling the condition. As this probability is based on the observations in a trace it is only an estimation of the true probability.

```
P(<working set> , <condition>)
```

Syntax 2.3 P function

In the first version of the PPL grammar [14] the working set was not part of the syntax for P. Not knowing what set to work from could make the condition ambiguous. Hence this change in P was made. For the same reason it is required that an instance variable is specified for the working set using the instance operator.

```
P(t(i), <condition>)
```

Example 2.5 working set

What P does is to take each instance *i* of the working set and check it against the condition. The condition is one or more relational expressions containing properties for instances of tasks. If there are

several expressions these are combined with *logic operators*. The probability is calculated by dividing the size of the subset of instances that fulfil the condition with the size of the working set.

2.5 Probes

As mentioned in Section 2.2 the traces also contain probe observations. One typical example of such an observation could be the size of a message queue, where a change in the size of the queue becomes a probe event in the trace. As also mentioned in Section 2.2 this observed value can be accessed as a data member of a task instance. But message queues are shared between different tasks. Thus it might be of value to check properties on the queue for several tasks. Furthermore looking at a probe as a data member of an instance only gives the value of the probe at the start of that instance. These values might not be representative for the probes. This only shows what values is changed to, and nothing about how long the probe has each value. During most of the time the probes might have values that are not changed to frequently. It could for example be that a message queue is empty most of the time. If this queue then has values perhaps just at the start of several tasks it would give the false impression that this queue is rarely empty. To get around these problems there is an option to look at the value of the probe over time rather than at the start time of task instances. The wildcard character '*' is used to represent the entire time of the trace as a set. This is a set of time units similar to how a task is a set of instances. From this set only a probe data member can be accessed. A P function with this set as the working set calculates the probability, not from how many instances fulfil the condition, but from how many time units the condition is fulfilled.

In Example 2.6 probe18 is used as a data member to state that it should always have a value greater than 0 at the start of task t. By changing the working set to *, the query will state that probe18 should be greater than 0 not only when t start but at all times as shown in Example 2.7.

$$P(t(i), t(i).probe18 > 0) = 1$$

Example 2.6 probe as data member

$$P(*,*.\ probe18 > 0) = 1$$

Example 2.7 probe over time

2.6 Operators

The expressions that make up the condition of a P function may contain arithmetic operations. The four basic arithmetic operators '+', '-', '*' and '/' can be used. They have their common use and precedence; '*' and '/' before '+' and '-'. Unary minus and parenthesises can also be used. Any arithmetic expression that can be constructed using these components is valid. The operators are applied to the data members of instances of tasks, numeric constants and statistical functions. The absolute value function *abs*, which takes an arithmetic expression as its only argument, can also be used.

In Example 2.8, a valid, albeit unnecessarily complex, PPL query stating that the response time of t should be less than 28 with a probability of less than 0.5.

```
P(t (i), t (i).resp < (abs(1 + 6) * 4) / (-1 + 2)) < 0.5
```

Example 2.8 arithmetic's as a constant

In Example 2.9 *abs* is used to formulate the property that the instances of t should start within 10 time units of any instance of s with a probability greater than 0.5. What task start first is not of interest, only that the difference in starting times is less than 10.

```
P(t(i), abs(t(i).start – s(j).start) < 10 ) > 0.5
```

Example 2.9 separation using abs

In Example 2.10, the probability of t meeting its deadline of 10 time units with a margin of 1 time unit should be greater than 0.5.

```
P(t(i), t(i).resp + 1 < 10) > 0.5
```

Example 2.10 arithmetic's on data member

PPL have the three logic operators *AND*, *OR* and *NOT*. All three are applied to relational expressions. AND and OR connects expressions that make up the condition of a P function. NOT is unary and gives the inversion of the expression it is applied to. Generally the NOT operator could be replaced by changing the relation operator of the expression. The expression in question might however be quite complex with several relations making it easier to use NOT than to translate it. AND and OR has the same precedence and NOT has higher precedence than them.

In Example 2.11 the AND operator is used to state that the response time of t should be between 5 and 10 time units, with a probability greater than 0.5.

```
P(t(i), t(i).resp > 5 AND t(i).resp < 10) > 0.5
```

Example 2.11 AND operator

In Example 2.12 the NOT operator is used to state that the response time of t should not be greater than 5, with a probability greater than 0.5. This query could be formulated without the NOT operator as in Example 2.13.

```
P(t(i), NOT(t(i).resp > 5)) > 0.5
```

Example 2.12 NOT operator

```
P(t(i), t(i).resp <= 5) > 0.5
```

Example 2.13 not greater than 5 without NOT operator

PPL has five relational operators, greater than '>', less than '<', greater than or equal '>=', less than or equal '<=' and strict equal '='. These operators can be applied to probabilities or numeric expressions. The use of these operators is twofold. If one of their operands is an unbounded variable then they become more of assignment operators than relational operators. If no unbounded variable is involved they have their regular relational meaning. All P functions contain at least one relational operation.

2.7 Unbounded variables

PPL queries may contain one unbounded variable. This variable can be used to return values to the user. For example it is possible to find a deadline or the probability of some property. Normally a query would answer only *true* or *false*. The unbounded variable can be part of the condition in a P function or as one operand in the outer relational operation of the query. An unbounded variable may not be part of an *arithmetic expression*. To allow them to be part of arithmetic expressions as well as allowing more then one unbounded variable was not desired features. Furthermore, those features would have been difficult to implement and thus the restrictions were made. The restriction on arithmetic's does not limit the language. Any arithmetic's could be rewritten to be applied to the expression the unbounded variable is compared to instead, e.g. t(i).resp = 2X can be written as t(i).resp/2 = X. Another restriction is that the unbounded variable may not be part of the argument to a *statistical function*. A statistical function is evaluated to a single value. That can not be done if it contains an unknown, i.e. an unbounded variable.

In Example 2.14 an unbounded variable, X, used in the condition of the query: what deadlines are met with a probability of at least 0.5.

```
P(t(i), t(i).resp <= X) >= 0.5
```

Example 2.14 inner unbounded variable

In Example 2.15 the unbounded variable, X, is used as the probability of the task t not meeting a deadline of 5.

$$P(t(i), t(i).resp > 5) = X$$

Example 2.15 outer unbounded variable

2.8 Statistical functions

There are four statistical functions, *min*, *max*, *avg* and *median* that can be used in the conditions of P functions. These functions are also allowed to be written as stand alone queries. These are all classic statistical functions. Each function has two versions. The first one take only one argument, the set they are to be applied to, i.e. the working set, as in Syntax 2.4. Unlike for the P function the working set for the statistical functions must also contain what data member of the task that should be used. The second version, Syntax 2.5, takes two arguments, a set and a condition. This second version does the same thing as the original but is applied only to the subset of the set that fulfil the condition.

```
<function>(<task>.<data member>)
```

Syntax 2.4 statistical function

```
<function>(<task>(<instance variable>).<data member> , <condition>)
```

Syntax 2.5 statistical function on a subset

Functions	
min()	Returns the smallest value of the set.
max()	Returns the greatest value of the set.
avg()	Returns the average of all the values in the set.
median()	Returns the median, the middlemost value of the set. If the size of the set is even then the median is calculated as the average of the two middlemost values.

Figure 2.2 The statistical functions of PPL

In Example 2.16 max is used in a P function to state that the response time of t should be less than the greatest response time observed for s with a probability of less than 0.5.

$$P(t(i), t(i).resp < max(s.resp)) < 0.5$$

Example 2.16 max used in a P

Example 2.17 shows a stand alone function used to find the average response time for the task t.

```
avg(t.resp)
```

Example 2.17 stand alone avg

In Example 2.18 a condition is used to get the average response time for the instances of t which have a start time greater than 5.

```
avg(t(i).resp , t(i).start > 5)
```

Example 2.18 stand alone avg on a subset

2.9 Following

With the instance operator it is possible to compare the same instance of different tasks, t(i) with s(i). However t(i) and s(i) might not have any connection, t(20) and s(20), for example, might be very far from each other in time. Such comparisons will likely be pointless unless t and s have the same rate. More useful would be to compare instances with similar start times, e.g. to compare t(i) with the next instance of s starting after t(i). To easily find instances of different tasks that are sequential in time like this PPL contain the function *following*. This function is used in the instance operator to map instances over time. Following takes, as its only argument, a task with an instance operator, e.g. t(i). What it does is to find the instance, of the task whose instance operator it is part of, that start execution closest after the end of the instance given as an argument.

Accessing data members of a task using the following function would be written as in Syntax 2.6.

```
<task>(following(<task>(<instance variable>))).<data member>
```

Syntax 2.6 following

In Example 2.19 all instances of s is compared to the instance of t closest after.

```
s(i).resp > t(following(s(i))).resp
```

Example 2.19 comparing s to the following t

Like with the plain instance operator plus and minus can be used to get relative instances. These arithmetic's can be applied to the instance variable in the argument, the result returned from the function or both.

t(following(s(i + 1)))

Example 2.20 the instance of t following the next instance of s

t(following(s(i)) + 1)

Example 2.21 the next instance of t after the instance of t following this instance of s

t(following(s(i+1)) + 1)

Example 2.22 the next instance of t after the instance of t following the next instance of s

Consider the execution Example 2.23, t(0) executing at time 2 and 4, s(0) at time 1, s(1) at time 3 etc where t(0) is pre-empted by s(1) and t(1) is pre-empted by s(3). Mapping instances of t following an instance of s would give the mappings presented in Example 2.24. As this shows, several instances can be followed by the same instance, e.g. both s(1) and s(2) is followed by t(1). It also shows that all instances are not necessarily followed. Here there is no instance of t after s(6). That will result in some instances being excluded from the query as explained in Section 3.7. Mapping instances of s following an instance of t would give mappings as shown in Example 2.25. Notice that s(2), not s(1), is following t(0) since it is the instance after the *end* of t(0).

t(i)		0		0				1		1	1						2			
s(i)	0		1			2			3				4		5			6		
time	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Example 2.23 an example trace

s(i)	t(following(s(i)))
0	0
1	1
2	1
3	2
4	2
5	2
6	-

Example 2.24 results from following

t(i)	s(following(t(i)))
0	2
1	4
2	6

Example 2.25 results from following

2.10 Instance sequencing

Common PPL queries concern properties over sequences of instances, e.g. a query like the probability of a number of consecutive instances all having a response time greater than some value. This would be done using the instance operator and the AND operator. Such a query could be quite large, especially if the property to be checked is a complexly formulated one. A mean to specify that a property should be true for several consecutive instances could be useful. Hence there is a feature of sequencing in the instance operator. Previously one integer was allowed to be added to the instance variable to compare sequences. Now a range of integers can be added to specify that not only should a property be true for the instance *i* but also for *i* + 1 and *i* + 2 and *i* + 3 etc.

Syntax 2.7 shows the syntax for the instance operator using sequencing. The sequence is written as the first and last number of the range within brackets, separated by '..'. It is then applied to the instance variable using one of the arithmetic operators '+' or '–'.

```
(<instance variable><aritop>[NUM..NUM])
```

Syntax 2.7 instance operator with sequencing

The property that three consecutive instances of t should have a response time greater than 5 with a probability greater than 0.5 is formulated in Example 2.26 using sequencing.. Example 2.27 show the same property formulated using AND operators instead of a sequence.

P(t(i), t(i + [0..2]).resp > 5) > 0.5

Example 2.26 comparing a range of instances using sequencing

P(t(i), t(i+0).resp > 5 AND t(i + 1).resp > 5 AND t(i + 2).resp > 5) > 0.5

Example 2.27 comparing a range of instances without sequencing

Sequencing can be combined with the function *following*. The sequence could be applied in the argument of *following* as in Example 2.28 stating that three consecutive instances of s should be followed by instances of t with a response time greater than 5 with a probability greater than 0.5. In Example 2.29 the sequence is applied to the result from *following*. There the query state that the probability of an instance of s being followed by three consecutive instances of t with response times greater than 5, should be greater than 0.5.

P(s(i), t(following(s(i + [0..2]))).resp > 5) > 0.5

Example 2.28 sequencing the argument to following

P(s(i), t(following(s(i)) + [0..2]).resp > 5) > 0.5

Example 2.29 sequencing the result from following

Example 2.30 is a combination of Example 2.28 and Example 2.29. It states that the probability, of three consecutive instances of s each being followed by three consecutive instances of t with response times greater than 5, should be greater than 0.5. Example 2.31 is similar to Example 2.30 with the difference that, instead of having response times greater than five all those instances should have response times greater than the response times of three instances of s.

P(s(i), t(following(s(i + [0..2])) + [0..2]).resp > 5) > 0.5

Example 2.30 sequencing both argument and result of following

P(s(i), t(following(s(i + [0..2])) + [0..2]).resp > s(i + [0..2]).resp) > 0.5

Example 2.31 comparing two sequences

2.11 Subset

The function *subset* is not used in expressions, instead it is used as a stand alone function to print a set of values to a file. This could, for example, be used to see all response times for some task. Like the statistical functions its first argument is the working set it should be applied to. The second, optional, argument is the condition if only a subset should be printed. Hence the syntax is quite similar to those functions. The difference is that after the function the operator '>' is used as a pipe to the file the result should be written to. When '*' is used as the task not only the values of the probe but also how many time units that value was held is written to the file.

```
subset(<task>.<data member>) > "<file>"
```

Syntax 2.8 the subset function without a condition

```
subset(<task>(<instance variable>).<data member>,<condition>) > "<file>"
```

Syntax 2.9 the subset function with a condition

```
subset(t(i).resp, t(i).resp > 5) > "t_resp.txt"
```

Example 2.32 writing response times greater than 5

3 Semantics of PPL

Here we present the semantics for PPL. As tasks are sets of instances we present the semantic rules in terms of set theory. Throughout this section we use the following notations. A capital letter represents a set, e.g. X. A set must contain instances, i.e. we do not allow the empty set in our queries. A lower case letter represents a constant or a variable. To distinguish between the two we denote constants with a c, e.g. x_c, and variables with a v, e.g. x_v. We use two labels to represent different operators; *aritop* is an arithmetic operator and *relop* is a relational operator. The meaning of a query will vary depending on if certain expressions are right or left operand of certain operators. To simplify the semantics we only explain them assuming this given form on the query. The P function is always the left operand of the outer relation. When an unbounded variable is used it is always the right operand of the relation it is part of. Hence using for example the greater than operator in Section 3.5, unbounded variables, means "greater than the unbounded variable", e.g. $3 > x_v$. As we for some semantic rules need to reason about relative order between instances we use the relative order operator, defined as follows in definition21 in [14]:

$x <^n y$ *is the relative order relation between* x *and* y *such that* $n-1$ *individuals are ordererd in between* x *and* y:

$x <^1 y$ *iff* $\neg\exists z : x < z < y$

$x <^2 y$ *iff* $\exists z : x <^1 z <^1 y$

$x <^n y$ *iff* $\exists z_1 \ldots z_{n-1} : x <^1 z_{n-1} <^1 \ldots <^1 z_1 <^1 y$

Especially we say that $x \in X$ *and* $y \in Y$ *have the same order in X respectively Y if* $x <^0 y$.

Definition 3.1 the relative order operator

3.1 Probability function P

The foundation of PPL is the probability function P. P can be defined as the probability of a set having some property. P takes two arguments. The first one is the working set. The second argument is the condition. What the function does is to create a subset containing all instances of the set that fulfil the condition. The size of this subset is then divided by the size of the working set to get a probability.

The working set must be given as an argument to avoid ambiguity. This ambiguity could also have been avoided with semantic rules explaining the expression. That however would more or less limit the flexibility or power of the query. The first set of the expression could for example have been considered the working set. Another solution would be to consider the working set the intersecting instances of all sets, i.e. the smallest set is the working set. These would all be inferior to the choice of explicitly stating the working set as an argument. This also makes the purpose of the query clearer. All queries would not be ambiguous without the working set, but for some it makes a difference,

especially when the query contains several different tasks. Consider Example 3.1 and Example 3.2, both have the same condition, the instance of X start during the execution of some instance of Y. Depending on what set is chosen as the working set the meaning, and result, of the query will vary.

In Example 3.1, using X(i) as the working set, P gives the probability of X pre-empting Y.

P(X(i), X(i).start > Y(j).start AND X(i).start < Y(j).end)

Example 3.1 X pre-empting Y

In Example 3.2 the working set is changed to Y(j). Here P gives the probability of Y being pre-empted by X.

P(Y(j), X(i).start > Y(j).start AND X(i).start < Y(j).end)

Example 3.2 Y being pre-empted by X

3.2 Instances

The instance operator assigns instance variables to tasks. Instance variables are split into two categories, *bounded instance variables* and *unbounded instance variables*. In Section 3.1, the probability function P was defined as the probability of the instances of its set fulfilling the condition. The working set was defined as a task with an instance operator. The instance variable in this operator is the one and only bounded instance variable of the query. When evaluating a query every possible binding for the bounded instance is evaluated. All other instance variables in the query are unbounded. For them it is not always necessary to try every possibility. Instead they are only bounded such that the condition is fulfilled. Consider Example 3.1 from above, in this query *i* is the bounded instance variable while *j* is unbounded. For every *i* a value on *j* should be found that makes the condition true. If such a *j* is found then this instance X(i) is considered true. If every possible *j* is tried without finding one that makes it true then that X(i) is false. If there are several *j* that would make it true makes no difference. What is asked for is the probability of X pre-empting some instance of Y, not how many Y or a specific instance of Y.

3.3 Arithmetic and relational operators

Three different basic constructions for arithmetic and relational operators are allowed in PPL, between two scalar values, between a set and a scalar or between two sets. A scalar value is a constant, the result from an arithmetic expression or the result from a statistical function. An arithmetic operation between two scalars is done by applying the operator to the two values.

$$x_c \text{ aritop } y_c \quad \Rightarrow \quad \{x_c \; aritop \; y_c\}$$

Semantic rule 3.1 arithmetic's between scalar values

The same is true for relational operations. The operator is simply applied to the two operands. However a relation between two constants does not contain any set. Thus it cannot be matched with instances of the working set and has a probability of either 0, if the relation is false or 1, if the relation is true.

$$P(X(i),\ x_c\ relop1\ y_c)\ relop2\ z_c \quad \Rightarrow \quad \begin{cases} true & if\ (x_c\ relop1\ y_c)\ relop2\ z_c \\ false & otherwise \end{cases}$$

Semantic rule 3.2 relation between scalar values

An operation between a set and a scalar is almost equally straightforward. The operation is applied to each instance of the set.

$$X(i)\ aritop\ x_c \Rightarrow \quad \{x\ aritop\ x_c : x \in X\}$$

Semantic rule 3.3 arithmetic's between a set and a scalar

For relational operations this gives a subset of instances that fulfil the condition. The size of this subset is divided by the working set to get the probability.

$$P(X(i),\ X(i)\ relop1\ x_c)\ relop2\ y_c \quad \Rightarrow \quad \begin{cases} true & if\ \dfrac{|\{x : x \in X \wedge x\ relop1\ x_c\}|}{|X|}\ relop2\ y_c \\ false & otherwise \end{cases}$$

Semantic rule 3.4 relation between a set and a scalar

There are three different kinds of operations between two sets. The variation depends on the instance variable in the instance operators of the sets, those between two bounded instance variables, between a bounded and an unbounded and between two unbounded. With two bounded the operation is applied index-wise, e.g. X(0) + Y(0), X(1) + Y(1), etc.

$$X(i)\ aritop\ Y(i) \quad \Rightarrow \quad \{x\ aritop\ y : x \in X \wedge y \in Y \wedge x <^0 y\}$$

Semantic rule 3.5 arithmetic's between two sets

$$P(X(i), X(i)\ relop1\ Y(i))\ relop2\ y_c \Rightarrow$$
$$\begin{cases} true & if \dfrac{\left|\left\{\langle x,y\rangle : x \in X \wedge y \in Y \wedge x\ relop1\ y \wedge x <^0 y\right\}\right|}{\left|\left\{\langle x,y\rangle : x \in X \wedge y \in Y \wedge x <^0 y\right\}\right|} relop2\ y_c \\ false & otherwise \end{cases}$$

Semantic rule 3.6 relation between two set with bounded instance variables

With these operations there is a risk that the cardinality of the sets is not the same. If that is the case some instances might be ignored. The goal is to try for every instance in the working set of the P. If the other sets using the bounded instance variable in their instance operator have more instances then those are ignored. Those extra instances are not of any interest in this case. If one of those set have less instances than the working set then there is no option but to ignore some of the instances in the working set as it is not possible to compare with something that does not exist. However, unlike the previous case, here those ignored instances will influence the result. Those instances that cannot be compared will be excluded from the query. Consider Example 3.3 stating that all instances of X should have a response time greater than their counterpart in Y. If X.resp = {3,4,3,2,4} and Y.resp = {3,2,4} then there is a cardinality problem as X has two instances more than Y. The effect will be that the last two instances of X will not be part of the query. Out of the three instances that is compared, only one, X(1) > Y(1); 4 > 2, fulfil the condition. Since two instances from X was excluded those cannot be taken into account when calculating the probability as shown in Example 3.4. The result from the P function will thus be one third, or 0.333. The answer for the query is false, 0.333 is not equal to 1.

$$P(X(i), X(i).resp > Y(i).resp) = 1$$

Example 3.3 comparing same instance in two sets

$$\frac{\left|\left\{\langle 4,2\rangle\right\}\right|}{\left|\left\{\langle 3,3\rangle,\langle 4,2\rangle,\langle 3,4\rangle\right\}\right|} = \frac{1}{3} = 0.333$$

Example 3.4 calculating result with cardinality problem

Operations between a bounded and an unbounded are a bit different. As explained in Section 3.2 an unbounded instance variable is bounded such that, if possible, the condition will be fulfilled. The operation is then applied between the bounded and one binding for the unbounded.

$$\text{X(i) aritop Y(j)} \quad \Rightarrow \quad \{ x \; aritop \; y : \forall x \wedge \exists y \wedge x \in X \wedge y \in Y \}$$

Semantic rule 3.7 arithmetic's between a set with a bounded instance, i, and a set with a unbounded instance, j

$$\text{P(X(i), X(i) relop1 Y(j)) relop2 } y_c \Rightarrow \begin{cases} true & if \; \dfrac{\left| \{ x : x \in X \wedge y \in Y \wedge x \; relop1 \; y \} \right|}{|X|} \; relop2 \; y_c \\ false & otherwise \end{cases}$$

Semantic rule 3.8 relation between a set with bounded and a set with unbounded instance

Example 3.5 states that all instances of X should have a response time greater than the response time of an instance of Y. As it says nothing about what instance of Y it should compare with each X could be compared with any Y. Thus there can not be any cardinality problem as several instances of X could be compared to the same instance of Y. If an instance of X has a response time greater than one or more of the instances of Y then it fulfil the condition. Assuming the same sets as above, X.resp = {3, 4, 3, 2, 4} and Y.resp = {3, 2, 4}, then four out of the five instances of X fulfil the condition. Only X(3) does not as its response time of 2 is not greater than any of the response times of Y. The two instances with response time 3, X(0) and X(2), is greater than 2, the response time of Y(1). The two instances with response times of 4, X(1) and X(4), is greater than both 3 and 2, Y(0) and Y(1), that however is no different from them being greater than only one. As long as they are greater than at least one, the condition is fulfilled. As Example 3.6 shows the result from the P function will be 0.8.

P(X(i), X(i).resp > Y(j).resp) = 1.

Example 3.5 relation between a set with bounded and a set with unbounded instance

$$\frac{\left| \{ \langle 3,2 \rangle, \langle 4,3 \rangle, \langle 3,2 \rangle, \langle 4,3 \rangle \} \right|}{|X|} = \frac{4}{5} = 0.8$$

Example 3.6 calculating result for a relation between a set with bounded and a set with unbounded instance

Example 3.7 is the same query as in Example 3.5 only with the working set changed from X(i) to Y(j). Using the same sets X.resp = {3, 4, 3, 2, 4} and Y.resp = {3, 2, 4} as above then two out of the three instances fulfil the condition, X(1) > Y(0) and X(0) > Y(1), as shown in Example 3.8.

$$P(Y(j), X(i).resp > Y(j).resp) = 1.$$

Example 3.7 relation between a set with bounded and a set with unbounded instance

$$\frac{\left|\left\{\langle 4,3\rangle, \langle 3,2\rangle\right\}\right|}{|Y|} = \frac{2}{3} = 0.666$$

Example 3.8 calculating result for a relation between a set with bounded and a set with unbounded instance

Between two unbounded instance variables is not much different from between a bounded and an unbounded. The difference is that if the query is to make any sense it must also contain another expression with the bounded instance variable. A P with only unbounded instance variables in the condition is like one with only constants. Their bindings will be the same for all instances of the working set. Thus the condition will be true for either all or none, i.e. the probability will be either 1 or 0.

$$X(j)\ aritop\ Y(k) \quad \Rightarrow \quad \{\, x\ aritop\ y : \exists x \wedge \exists y \wedge x \in X \wedge y \in Y \}$$

Semantic rule 3.9 arithmetic's between two sets with unbounded instance variables

$$P(X(i), X(j)\ relop1\ Y(k))\ relop2\ y_c \Rightarrow \begin{cases} true & if\ \ (x\ relop1\ y : x \in X \wedge y \in Y)\ relop2\ y_c \\ false & otherwise \end{cases}$$

Semantic rule 3.10 relation between two sets with unbounded instance variables

Using arithmetic's in the instance operator will quite possibly lead to comparing sets with different cardinality. For example comparing X(i) with X(i+1), X(i+1) will be a subset of X(i) containing all but the first instance. The first instance of X(i+1) will be X(0+1), i.e. X(1). Consequently there will be no instance for the last instance of X(i) to be compared with as there are no instance X(n+1) if X has n instances. To deal with this that last instance has to be excluded. Depending on the constant added to i several trailing instances could be excluded. For example comparing X(i) with X(i+5) the last five instances of X(i) would not be compared. If the arithmetic operator is '-' instead of '+' would give the same problem only then it is the first instances of X(i) that is excluded. Naturally this cardinality

problem could also occur when comparing different sets. As discussed previously in this section this can happen even without applying arithmetic's as the two sets can have a different number of instances. The problem is solved like for those without arithmetic's in the instance operator, Semantic rule 3.6, with the exception that it is now for $x <^{n} y$ instead of $x <^{0} y$.

$$\text{P(X(i), X(i) relop1 Y(i aritop n)) relop2 } y_c \Rightarrow \begin{cases} true & if \ \dfrac{\left|\left\{\langle x, y\rangle : x \in X \wedge y \in Y \wedge x \ relop1 \ y \wedge x <^{n} y\right\}\right|}{\left|\left\{\langle x, y\rangle : x \in X \wedge y \in Y \wedge x <^{n} y\right\}\right|} \ relop2 \ y_c \\ false & otherwise \end{cases}$$

Semantic rule 3.11 relation with arithmetic's in the instance operator

These cardinality problems can never occur for unbounded instance variables. However if *j* is an unbounded instance variable then adding *n* to it would eliminate the *n* first possible bindings as the first one tried would be (0+n).

3.4 Probes

The condition of a P on probes, a P with the working set *, can contain all the constructions that those with tasks have. Except for the instance operator as there is no instances here. The only difference is that instead of calculating a probability based on instances it is calculated on time units. A relational operation on a probe, the '| |' operator here count the number of time units rather than elements in a set. That is, the number of time units that probeX fulfil the condition divided with the total number of time units that probeX have a value.

$$\text{P(*, *.probeX relop1 } x_c \text{) relop2 } y_c \Rightarrow \begin{cases} true & if \ \dfrac{\left|probeX \ relop1 \ x_c\right|}{\left|probeX\right|} \ relop2 \ y_c \\ false & otherwise \end{cases}$$

Semantic rule 3.12 relation in a P on probes

The semantics for using a probe as a data member is no different compared to for any other data member. The probability is calculated as the number of instances of the working set that fulfils the condition divided with the total number of instances in the working set. The difference when using a probe as data member is that the value of the probe can not be found directly. Instead the probe event in the execution trace, whose value should be used, need to be found. The value of the probe data member should be the value of the probe when the task instance starts, i.e. the value of the probe event with the greatest timestamp that is less than the start time of the task instance.

$$\text{X(i).probeX} \Rightarrow \quad \text{X(i).probeX} = \begin{cases} \{e.value : e.time = t \wedge e \in probeXevents\} \\ t = \max(\{e.time : e.time < X(i).start \wedge e \in probeXevents\}) \end{cases}$$

Semantic rule 3.13 the value of a probe data member

3.5 Unbounded variables

A query may contain one unbounded variable. The unbounded variable decides the purpose of the query. If there is no unbounded variable then the query decides if a statement about the relation between two probabilities is true or not. If the query contains an unbounded variable then its purpose is to bind the variable to such values that the query is true. Hence the relational operators, when applied to an unbounded variable, function more like assignments than traditional comparisons. The unbounded variables can be used in two contexts. As a probability or inside the condition of a P. Queries with unbounded probabilities are relatively straightforward. First the P is evaluated and then an interval based on the result is assigned to the variable. For strict equal the result is not an interval. Instead the variable is bounded to the probability from P.

$$\text{P(X(i), X(i) relop } x_c \text{) = } y_v \qquad \Rightarrow \qquad y_v = \frac{|\{x : x \in X \wedge x \; relop \; x_c\}|}{|X|}$$

Semantic rule 3.14 outer unbounded variable with outer strict equal

For less than and less than equal the probability from P is the highest value of the interval. The lowest possible probability is 0. The two operators differ in that for less than equal the high value should be included in the interval while for less than the interval is only up to that value.

$$\text{P(X(i), X(i) relop } x_c \text{) < } y_v \qquad \Rightarrow \qquad y_v = [0..x_m) : x_m = \frac{|\{x : x \in X \wedge x \; relop \; x_c\}|}{|X|}$$

Semantic rule 3.15 outer unbounded variable with outer less than

$$\text{P(X(i), X(i) relop } x_c \text{) <= } y_v \qquad \Rightarrow \qquad y_v = [0..x_m] : x_m = \frac{|\{x : x \in X \wedge x \; relop \; x_c\}|}{|X|}$$

Semantic rule 3.16 outer unbounded variable with outer less than equal

Greater than and greater than equal work like the less than operators with the exception that the result from the P is the lowest value of the interval and not the highest.

$$P(X(i), X(i) \text{ relop } x_c) > y_v \quad \Rightarrow \quad y_v = (x_m..1] : x_m = \frac{|\{x : x \in X \wedge x \; relop \; x_c\}|}{|X|}$$

Semantic rule 3.17 outer unbounded variable with outer greater than

$$P(X(i), X(i) \text{ relop } x_c) >= y_v \quad \Rightarrow \quad y_v = [x_m..1] : x_m = \frac{|\{x : x \in X \wedge x \; relop \; x_c\}|}{|X|}$$

Semantic rule 3.18 outer unbounded variable with outer greater than equal

The unbounded variable must not necessarily be a probability. It could also be part of an expression. Finding bindings for these variables are more difficult than for the unbounded probabilities. The unbounded variables are bounded to intervals. These intervals can be constructed by finding thresholds. A threshold is a value that changes the result for the P. These thresholds are found around the values that the expression the unbounded variable is compared to can take. For example if the variable is compared with the response times of X, X(i).resp < x_t. If X.resp = {3, 4, 2} then those values are 3, 4 and 2. When solving these queries the query should be evaluated with the unbounded variable bounded to every one of those values. From this set of possible bindings a subset is gathered with all the bindings that makes both the outer and inner relation true. Depending on the operators the max or min value of this subset is taken to create an interval of valid bindings, i.e. bindings that make the query true. If the min or the max value should be used depends on the combination of operators. For some combinations just min and max is not enough as the interval should be up, or down, to the value that is just outside the interval. This is the case for example when both operators is '>'. There the interval is from -8 up to the smallest invalid binding, i.e. the smallest of the bindings that make the query false. In the semantic rules below the set of invalid bindings is created as the complement of the set of valid bindings. If the start or end of an interval is the least or greatest value in a set of possible bindings then the values just outside of this set must also be tested. If the value just outside this set is also a valid binding then the start/end of the interval should be -8/8 instead. Consider Example 3.9 with the set X.resp = {3, 4, 2}. All three values are valid binging so the value just outside the set must also be tested. For the operators in this example the interval should end with the max value. Thus the extra value to test will be 5. 5 is not a valid binding meaning that the unbounded variable will be bounded to the interval (-8..4]. Consider the similar Example 3.10 with the same set. As in Example 3.9 all three values are valid and the extra value 5 must be tested. In this case however, 5 is also a valid binding and the unbounded variable will be bounded to (-8..8).

$$P(X(i), X(i).resp < x_v) <= 0.8$$

Example 3.9 inner unbounded variable

$$P(X(i), X(i).resp < x_v) <= 1$$

Example 3.10 inner unbounded variable

When the inner operator is not strict equal there will be no difference in the assign method between an outer '<' and '<=' or between a '>' and '>='. Hence in the following eight rules any outer '<' could be replaced with a '<=' and any outer '>' could be replaced with a '>='.

$$P(X(i), X(i) < y_v) < x_c \quad \Rightarrow \quad y_v = \left(-\infty..\max(S)\right] : S = \left\{ x : x \in X \wedge \frac{\left|\left\{ i : i \in X \wedge i < x \right\}\right|}{\left|X\right|} < x_c \right\}$$

Semantic rule 3.19 inner unbounded variable with inner less than and outer less than

$$P(X(i), X(i) > y_v) > x_c \quad \Rightarrow \quad y_v = \left(-\infty..\min(S)\right) : S = \left\{ x : x \in X \wedge \frac{\left|\left\{ i : i \in X \wedge i > x \right\}\right|}{\left|X\right|} > x_c \right\}^C$$

Semantic rule 3.20 inner unbounded variable with inner greater than and outer greater than

$$P(X(i), X(i) >= y_v) > x_c \quad \Rightarrow \quad y_v = \left(-\infty..\max(S)\right] : S = \left\{ x : x \in X \wedge \frac{\left|\left\{ i : i \in X \wedge i \geq x \right\}\right|}{\left|X\right|} > x_c \right\}$$

Semantic rule 3.21 inner unbounded variable with inner greater than equal and outer greater than

$$P(X(i), X(i) <= y_v) < x_c \quad \Rightarrow \quad y_v = \left(-\infty..\min(S)\right) : S = \left\{ x : x \in X \wedge \frac{\left|\left\{ i : i \in X \wedge i \leq x \right\}\right|}{\left|X\right|} < x_c \right\}^C$$

Semantic rule 3.22 inner unbounded variable with inner less than equal and outer less than

$$P(X(i),\ X(i) > y_v) < x_c \quad \Rightarrow \quad y_v = [\min(S)..\infty) : S = \left\{ x : x \in X \wedge \frac{\left|\{ i : i \in X \wedge i > x \}\right|}{|X|} < x_c \right\}$$

Semantic rule 3.23 inner unbounded variable with inner greater than and outer less than

$$P(X(i),\ X(i) < y_v) > x_c \quad \Rightarrow \quad y_v = (\max(S)..\infty) : S = \left\{ x : x \in X \wedge \frac{\left|\{ i : i \in X \wedge i < x \}\right|}{|X|} > x_c \right\}^C$$

Semantic rule 3.24 inner unbounded variable with inner less than and outer greater than

$$P(X(i),\ X(i) <= y_v) > x_c \quad \Rightarrow \quad y_v = [\min(S)..\infty) : S = \left\{ x : x \in X \wedge \frac{\left|\{ i : i \in X \wedge i \leq x \}\right|}{|X|} > x_c \right\}$$

Semantic rule 3.25 inner unbounded variable with inner less than equal and outer greater than

$$P(X(i),\ X(i) >= y_v) < x_c \quad \Rightarrow \quad y_v = (\max(S)..\infty) : S = \left\{ x : x \in X \wedge \frac{\left|\{ i : i \in X \wedge i \geq x \}\right|}{|X|} < x_c \right\}^C$$

Semantic rule 3.26 inner unbounded variable with inner greater than equal and outer less than

When using a strict equality in the query there is a risk of getting an undecidable expression. It is quite possible that there are not enough instances of a task to get some probabilities. Consider the query in Example 3.11, if X contains an odd number of instances then there is no possible condition that half of the instances meet. If X had for example 3 instances then 1.5 instances would have to fulfil the condition. As there are no such thing as half instances this expression is undecidable. The only times a query with strict equal is certain to be decidable is when the comparison is with either 1 or 0. There is always a value for x_v such that all or none of the instances fulfil the condition. For those expressions that are decidable the valid bindings are found in the same manner as for the other relational operators. When the outer operator is strict equal and the inner is any but strict equal then an interval will be assigned to the unbounded variable. Unlike for without strict equal the interval does not necessarily include infinity.

$$P(X(i),\ X(i)\ relop\ x_v) = 0.5$$

Example 3.11 a possibly undecidable query

For the operators '>' and '<=' the first value in the interval is the smallest valid binding. The interval then range up to the smallest of the invalid bindings that are greater than the valid bindings. If the bindings are sorted then this would mean that the start of the interval is the first valid and the end of the interval is the first invalid after that first valid.

$$P(X(i), X(i)\ relop\ y_v) = x_c \Rightarrow$$

$$y_v = [t_1..t_2): \begin{cases} t_1 = \min\left(\left\{x : x \in X \wedge \frac{\left|\{i : i \in X \wedge i\ relop\ x\}\right|}{|X|} = x_c\right\}\right) \\ t_2 = \min\left(\left\{x : x \in X \wedge \frac{\left|\{i : i \in X \wedge i\ relop\ x\}\right|}{|X|} = x_c\right\}^C\right) \\ t_1 < t_2 \end{cases}$$

Semantic rule 3.27 inner unbounded variable with inner greater than or less than equal and outer strict equal

For the operators '<' and '>=' it is the other way around. Assuming a sorted set of bindings the start of the interval is the invalid before the first valid. The end of the interval is the greatest valid binding.

$$P(X(i), X(i)\ relop\ y_v) = x_c \Rightarrow$$

$$y_v = (t_1..t_2]: \begin{cases} t_1 = \max\left(\left\{x : x \in X \wedge \frac{\left|\{i : i \in X \wedge i\ relop\ x\}\right|}{|X|} = x_c\right\}^C\right) \\ t_2 = \max\left(\left\{x : x \in X \wedge \frac{\left|\{i : i \in X \wedge i\ relop\ x\}\right|}{|X|} = x_c\right\}\right) \\ t_1 < t_2 \end{cases}$$

Semantic rule 3.28 inner unbounded variable with inner less than or greater than equal and outer strict equal

It is also possible to get undecidable expressions when having a strict equality as the inner relation, e.g. Example 3.12. It is not necessarily the case that X has enough instances with the same value to fulfil the condition. Consider for example the set {1, 2, 3, 4, 2} and the relational operator '>'. There is no value for x_v that more than half of the instances are equal to.

$$P(X(i),\ X(i) = x_v)\ relop\ 0.5$$

Example 3.12 a possibly undecidable query

When the outer operator is '<' or '<=' the unbounded variable can be bounded to any value but the ones from our set of bindings that are not valid. Hence the unbounded variable will not be bounded to one but several intervals. These are the intervals that are created when splitting the interval (-8..8) at the values from the subset of invalid bindings.

$$P(X(i),\ X(i) = y_v)\ relop\ x_c \quad \Rightarrow$$

$$y_v = (-\infty..S_0)(S_0..S_1)(S_1\ \ ...\ \ S_{n-1})(S_{n-1}..S_n)(S_n..\infty) : S = \left\{ x : x \in X \wedge \frac{\left|\{ i : i \in X \wedge i = x \}\right|}{|X|}\ relop\ x_c \right\}^C$$

Semantic rule 3.29 inner unbounded variable with inner strict equal and outer less than or less than equal

Example 3.13 would, if X = {1, 2, 3, 4, 2}, give y_v = (-8..2)(2..8). The probability of X(i) = 2 is 2 / 5 = 0.4. For any of the other values in the set the probability is 0.2. For any value not included in that set the probability is naturally 0. Thus the query is true for any value between -8 and 8 except 2.

$$P(X(i),\ X(i) = y_v) < 0.4$$

Example 3.13 inner unbounded variable with inner strict equal and outer less than

With the outer operators '>', '>=' and '=' the unbounded variable is no bounded to any interval at all. Instead the only values for the unbounded that makes the query true is exactly the valid bindings. Hence instead of an interval the set of valid bindings is assigned to the unbounded variable.

$$P(X(i),\ X(i) = y_v)\ relop\ x_c \quad \Rightarrow \qquad y_v = \left\{ x : x \in X \wedge \frac{\left|\{ i : i \in X \wedge i = x \}\right|}{|X|}\ relop\ x_c \right\}$$

Semantic rule 3.30 inner unbounded variable with inner strict equal and outer greater than or greater than equal

Consider Example 3.14, it is the same query as in Example 3.13 above only with a different outer operator. If X = {1, 2, 3, 4, 2} then X(i) = 2 gives the probability 0.4. Any other value gives 0.2 or 0. For this query that means y_v = [2]. 2 is the only value that gives a probability greater than or equal to 0.4.

$$P(X(i), X(i) = y_v) >= 0.4$$

Example 3.14 inner unbounded variable with inner strict equal and outer greater than equal

There is one exception to those two rules, when the outer relation is '= 0' Semantic rule 3.29 should be used, not Semantic rule 3.30 as normally when both are strict equal. The values to try for y_v are the values in the set X. Thus the condition is always true for at least one instance as all values are equal to themselves. Because of this it is not possible to get a probability of 0 when both operators are strict equal. However there are undeniably values that would result in a probability of 0. Any value that is not in the set X would give that. Hence X could be bounded to any value except the ones in the set X. That is exactly what is done in Semantic rule 3.29, and thus it can be used for this case.

As discussed previously in this section these queries can be undecidable. There are no restrictions on them in order to prevent this. There are two reasons for this. First to fully remove the risk of undecidable expressions they would need to be restricted to the point where strict equality may barely be used at all. The only certain queries are those where probabilities are compared to 1 or 0. Secondly it is easy to, when trying to bind the variable, find out if the expression is undecidable. If a query is found to be undecidable then the answer will simply be that there is no valid binding.

3.6 Logic operators

The logic operators AND and OR connect several expressions. Thus the semantics for a query with logic operators is a combination of the semantic rules for its expressions. With the AND operator the same kind of cardinality problems that was discussed previously, in Section 3.3, concerning instances can occur. If the same instance variable is used in instance operators for different sets in two expressions connected by AND then it might not be possible to compare all the instances. In the same way as before only as many instances as the smallest of the involved sets contain can be compared. Any other instances must be ignored.

$$P(X(i), X(i)\ relop1\ x_c\ \text{AND}\ Y(i)\ relop2\ y_c)\ relop3\ z_c \quad \Rightarrow$$

$$\begin{cases} true & if \dfrac{\left|\left\{\langle x,y\rangle : x \in X \wedge y \in Y \wedge x\ relop1\ x_c \wedge y\ relop2\ y_c \wedge x <^0 y\right\}\right|}{\left|\left\{\langle x,y\rangle : x \in X \wedge y \in Y \wedge x <^0 y\right\}\right|}\ relop3\ z_c \\ false & otherwise \end{cases}$$

Semantic rule 3.31 AND without unbounded instance variables

P(X(i), X(i) relop1 x_c AND Y(j) relop2 y_c) relop3 z_c $\Rightarrow$

$$\begin{cases} true & if \ \dfrac{\left|\{x : x \in X \wedge y \in Y \wedge x \ relop1 \ x_c \wedge y \ relop2 \ y_c\}\right|}{|X|} \ relop3 \ z_c \\ false & otherwise \end{cases}$$

Semantic rule 3.32 AND with unbounded instance variables

Expressions connected with the OR operator work similar to those connected with AND. The significant difference is that the cardinality problem not occurs. If the set in the other expression contain fewer instances, then that expression is simply considered to be false for those instances. This can be done since OR only require one of its operands to be true.

X	Y	X < 3	Y < 3	AND	OR
1	4	true	false	false	true
3	5	false	false	false	false
2	2	true	true	true	true
4		false			false
2		true			true

Example 3.15 truth table for two sets X and Y

P(X(i), X(i) relop1 x_c OR Y(i) relop2 y_c) relop3 z_c $\Rightarrow$

$$\begin{cases} true & if \ \dfrac{\left|\{x : x \in X \wedge y \in Y \wedge (x \ relop1 \ x_c \vee y \ relop2 \ y_c) \wedge x <^0 y\}\right|}{|X|} \ relop3 \ z_c \\ false & otherwise \end{cases}$$

Semantic rule 3.33 OR without unbounded instance variables

P(X(i), X(i) relop1 x_c OR Y(j) relop2 y_c) relop3 z_c $\Rightarrow$

$$\begin{cases} true & if \ \dfrac{\left|\{x : x \in X \wedge y \in Y \wedge (x \ relop1 \ x_c \vee y \ relop2 \ y_c)\}\right|}{|X|} \ relop3 \ z_c \\ false & otherwise \end{cases}$$

Semantic rule 3.34 OR with unbounded instance variables

Queries with unbounded variables and AND or OR operators work like the regular unbounded variable queries. Is it the probability that is unbounded then there is no difference at all as the variable is bounded to the result from the P function. In the other cases the AND or OR is added to the condition when the subset of valid bindings is created. Then the variable is, depending on the relational operators, bounded to this set or an interval involving the min or max of it, as previously defined in Section 3.5. As examples the semantic rules for when both relations are less than, i.e. logic operators added to Semantic rule 3.19.

$$P(X(i), X(i) < y_v \text{ AND } Y(i) \text{ relop } y_c) < x_c \Rightarrow$$
$$y_v = [-\infty..\max(S)] : S = \left\{ x : x \in X \wedge \frac{\left| \left\{ \langle i, y \rangle : i \in X \wedge y \in Y \wedge i < x \wedge y \; relop \; y_c \wedge i <^0 y \right\} \right|}{\left| \left\{ \langle i, y \rangle : i \in X \wedge y \in Y \wedge i <^0 y \right\} \right|} < x_c \right\}$$

Semantic rule 3.35 AND operator and inner unbounded variable with inner and outer less than

$$P(X(i), X(i) < y_v \text{ OR } Y(i) \text{ relop } y_c) < x_c \Rightarrow$$
$$y_v = [-\infty..\max(S)] : S = \left\{ x : x \in X \wedge \frac{\left| \left\{ i : i \in X \wedge y \in Y \wedge (i < x \vee y \; relop \; y_c) \wedge i <^0 y \right\} \right|}{|X|} < x_c \right\}$$

Semantic rule 3.36 OR operator and inner unbounded variable with inner and outer less than

If the operand to the NOT operator is *true* then the result from the operation is *false*. If the operand is *false* then the result will be *true*.

$$\text{NOT(exp)} \Rightarrow \begin{cases} true & if \text{ exp} = false \\ false & otherwise \end{cases}$$

Semantic rule 3.37 NOT operator

3.7 Following

The function *following* does not really add or change anything to the semantics of the query. Its purpose is only to map instances on their start times. The instance of Y that follows X(i) is the first instance of Y executing after the end of X(i).

$$\text{Y(following(X(i)))} \Rightarrow \quad Y(j) : Y(j).end = \min(\{y : y \in Y.end \wedge y > X(i).end\})$$

Semantic rule 3.38 following

The semantics for a query containing *following* is naturally the same as for those without. The only difference is the introduction of the risk that not all instances of X necessarily have a matching instance of Y. There might not be any instances in X that start after some instance of Y. An extreme example would be that first all instances of Y execute and then all instances of X execute. This is basically the same problem as the cardinality problems discussed in Section 3.3 and thus can be solved the same way. Those instances that do not have a match are excluded from the query.

3.8 Instance sequencing

Instance sequencing is a simplified way of writing expressions containing several AND operators. When several consecutive instances are to be compared they can simply be written as one comparison on a sequence instead of several comparisons combined with ANDs. The range of the sequence must be increasing. The numbers may be negative as for example looking at the previous, current and following instance, X(i + [-1..1]). As these queries are only a number of AND connected expressions they are solved no different than other queries with AND operators as defined in Section 3.6. The query is first translated from sequence to AND form, i.e. any sequence is replaced by several expressions connected by AND.

$$
\begin{aligned}
X(i\ aritop\ [n..m])\ relop\ x_c \quad \Rightarrow \quad & X(i\ aritop\ n)\ relop\ x_c\ AND \\
& X(i\ aritop\ n+1)\ relop\ x_c\ AND \\
& X(i\ aritop\ n+2)\ relop\ x_c\ AND \\
& \ldots \\
& X(i\ aritop\ m-1)\ relop\ x_c\ AND \\
& X(i\ aritop\ m)\ relop\ x_c
\end{aligned}
$$

Example 3.16 translating relation between sequence and scalar

The sequence does not have to be compared to a constant. It could also be compared with another sequence. Comparing two sequences results in a matrix like effect as each instance of the first sequence is compared to each instance of the second, i.e. the cross product.

X(i aritop1 [n..m]) relop Y(i aritop2 [a..b]) ⇒ X(i aritop1 n) relop Y(i aritop2 a) AND
X(i aritop1 n) relop Y(i aritop2 a+1) AND
…
X(i aritop1 n) relop Y(i aritop2 b) AND
X(i aritop1 n+1) relop Y(i aritop2 a) AND
X(i aritop1 n+1) relop Y(i aritop2 a+1) AND
…
X(i aritop1 n+1) relop Y(i aritop2 b) AND
…
X(i aritop1 m) relop Y(i aritop2 a) AND
X(i aritop1 m) relop Y(i aritop2 a+1) AND
…
X(i aritop1 m) relop Y(i aritop2 b)

Example 3.17 translating a relation between two sequences

Sequences may be combined with the function *following* in two ways. The instance given in the argument to *following* may contain a sequence as in Example 3.18. A sequence can also be added to the index returned from *following* as in Example 3.19. With the sequence given in the argument of the *following* function results in a number of *following* functions with different arguments when translating the queries to "AND-form". One for each instance in the sequence. When applied after the *following* function the argument or return value will not change. Instead the sequence is applied to the index returned. Moreover the two could be combined as in Example 3.20. This results in a sequence of indexes from *following*. To each of them the second sequence is applied.

X(following(Y(i aritop [n..m]))) relop x_c ⇒ X(following(Y(i aritop n))) relop x_c AND
X(following(Y(i aritop n+1))) relop x_c AND
…
X(following(Y(i aritop m))) relop x_c

Example 3.18 translating a sequence as argument to following

X(following(Y(i)) aritop [n..m]) relop x_c ⇒ X(following(Y(i)) aritop n) relop x_c AND
X(following(Y(i)) aritop n+1) relop x_c AND
…
X(following(Y(i)) aritop m) relop x_c

Example 3.19 translating a sequence applied to the result from following

X(following(Y(i aritop1 [n..m])) aritop2 [a..b]) relop x_c ⇒

X(following(Y(i aritop1 n)) aritop2 a) relop x_c AND
X(following(Y(i aritop1 n+1)) aritop2 a) relop x_c AND
…
X(following(Y(i aritop1 m)) aritop2 a) relop x_c AND
X(following(Y(i aritop1 n)) aritop2 a+1) relop x_c AND
X(following(Y(i aritop1 n+1)) aritop2 a+1) relop x_c AND
…
X(following(Y(i aritop1 m)) aritop2 a+1) relop x_c AND
X(following(Y(i aritop1 n)) aritop2 b) relop x_c AND
X(following(Y(i aritop1 n+1)) aritop2 b) relop x_c AND
…
X(following(Y(i aritop1 m)) aritop2 b) relop x_c

Example 3.20 translating a following with sequences both in argument and on its result

4 The tool

A tool has been implemented that evaluates PPL queries using the semantics defined in Section 3. The tool is divided into three parts. A parser that recognises queries and builds them into suitable tree structures, a log file compiler that reads a trace and compiles its data into task instances, and the actual query evaluator, that evaluates queries from the parser on the data from the compiler. The three parts are combined into one application.

The tool does not come with a graphical user interface. Although it can be used by itself it is intended to be run from some other application. It takes three arguments, a log file, a query file and a result file. The log file is the trace to be analysed. The query file contains one or more PPL queries. If the file contain several queries then they are separated by ';'. The result file is where the tool writes the results for the queries.

4.1 Parser

The first stage of the tool is to read the query file and check the queries for errors. For this a *scanner* and a *parser* was created using two tools: Flex [11] and Bison [7]. Flex creates a scanner given an input file containing the regular expressions it should recognize and what actions to take for each expression. Bison creates a bottom up parser given a context free grammar. In this grammar semantic actions are used to build an *Abstract Syntax Tree*, AST, of the query. These actions are also used to perform various error checking.

The scanner recognises all valid constructs that can be part of a query and creates tokens for them. Each token is passed on to the parser. If what it reads does not match any of the regular expressions for valid constructs then an error message is generated and scanning is aborted.

The parser calls the scanner to get the next token. If the token received match a following terminal in the grammar then that terminal is consumed and the parser continues with the next token. If no matching terminal is found then there is a parse error, a syntactical error in the query. If a parse error is found the parsing is aborted. The second purpose of the parser, the first being syntax checking, is to construct an AST from the query. At the end of each grammar rule a tree node is constructed and passed up to the rule above. Each node contains pointers to all of its sub trees plus other information like data or operator depending on what kind of node it is (see Appendix B for details on the nodes). The grammar also has semantic actions to check that no P function contain both the all tasks set, '*', and regular tasks.

If the query parsed contains any sequences then the next step after parsing is to translate all sequences into AND connected expressions. The translation is done by a recursive algorithm traversing the AST in post order. It searches for a node containing a sequence. On its way it stores the latest visited node for a logic expression and the latest visited node for a relational expression. Once a sequence node is found copies are made of the expression it is part of. This expression is the latest relational expression that is stored. One copy is made for each index in the sequence. The sequence node of the copy is replaced by a regular index node. All the copies are then connected with AND operators. The final

task is to connect all these nodes to the tree. This is where the stored latest logic expression node is used. That node is simply the point where these new nodes are to be inserted. Now that the sequence is translated the algorithm start over. From the root of the tree it searches for another sequence to translate. It cannot just continue from where it was as the expression it has made copies of might contain several sequences. Thus it needs to start from the beginning after each translation. The algorithm is done once it has traversed the entire tree without finding any sequence.

After sequences have been translated it is time for type checking. Type checking is also done by an recursive algorithm traversing the AST in post order. There are two types, NUM and BOOL. Unbounded variables are considered as type NUM since they are to be bounded to numeric values. On the way up from the recursion each node returns its type. For each operator node the types of the operand expression(s) is checked. If the type is wrong the type TYPE_ERROR is returned. For the property node, the root node of the tree that make up the outermost relation between two probabilities, a special check is performed to make sure that if an operand is a constant it must not be greater than 1 or less than 0. In addition such comparisons, i.e. comparing if a probability is greater than 1 or less than 0, is also not allowed. The next step is to do an "unbounded check". A query may never contain more than one unbounded variable. The entire AST is traversed and for every found unbounded variable node a counter is increased by one. If the value of this counter becomes greater than 1 then there are too many unbounded variables in the query.

4.2 Log file compiler

The log files that are to be analysed consist of three parts, first a header containing, among other things, the number of tasks in the system, then a task list with the names and id of all the tasks, and finally a list of events. These events are raw data about task switches and probe observations. In order to evaluate queries on this data the task switch events must first be compiled into task instances.

The purpose of this step is to create two lists, a probe list and a task list, for use in the query evaluation. The task list is to contain all tasks where each task has a list of all its instances. The probe list is similar. It contains all the probes where each probe has a list of all observations it has made. For each task instance data from several events are needed. At least two events are needed, one for the start and one for the end of the instance. Thus a temporary working instance is needed. As tasks pre-empt each other there will be times when information about more than one task instance need to be kept. Thus a list of such incomplete instances is needed. This is called the active list as it is the list of the tasks currently active. When a task switch event is read it gives information about the state of the previous task and what task was started. If the started task was not in the active list then it is added. Depending on the state of the previous task it is either done with its execution or waiting to execute once again. If the task is not done then its execution time is increased with the time it had been running since last started/resumed. The task is then left in the active list. If the task is done executing then the end time is set and its response time calculated as well. This task is then removed from the active list and added as an instance in its list in the task list. For probes there are no instances, only observed changes are stored. Thus there is no need to keep an active list for them. Instead a probe event is simply added to the event list for that probe.

A log file contains events from only a short period of the systems running time. Because of this it might be that some instances recorded are incomplete, i.e. they start or end outside of the log. Consider the example log in Figure 4.1. Task A is first noticed when it is switched to at time 2. The problem is that there is no way of knowing whether this is a new instance of A or if it was a resumption of an instance started outside of the log. From the log it is impossible to see any difference between a resumed and a started task. Thus it must be assumed that this is the start of a new instance risking the introduction of a slight error. Task B in the example start just before the beginning of the log and is the task running when the log begin. The first event of the log is task B being done and A switched in. The event does however not say that it was task B that was done. It only shows that the currently running task was done. Thus there is no way of knowing that it was task B. In this case there is no other option but to ignore that B. Task C was pre-empted by D, i.e. it is not yet done with its execution. But the remainder of its execution occurs after the end of this log file. In this case there are some options. Enough information has been gathered to construct a valid instance of C. But it is known that not all information has been gathered making this instance faulty. The two most straightforward solutions would be to either remove the instance as it is incomplete or to add the current part as an instance. Other ways would be to make some educated guess on how the remainder of the instance looks. No matter what option is chosen some fault is introduced. The first option, to remove it, was chosen reasoning that the lack of that instance is a lesser error than adding a faulty instance. Task D is the task running at the end of the log file. It is similar to B only here there is a start but no end. D is also similar to C in the way that an instance could be constructed assuming the end of the log file as the end of the instance. Like with C the option to remove the instance rather than adding incomplete information was chosen.

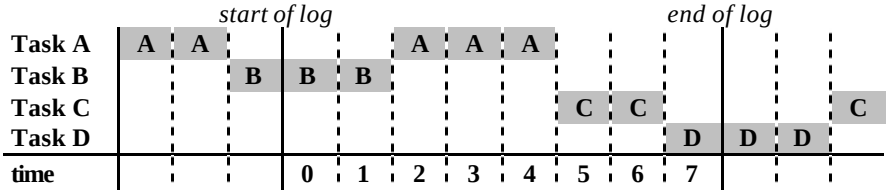

Figure 4.1 an example log

4.3 Query evaluation

There are two kinds of queries, those with properties and those that, consist of a function only. A function is simply evaluated the same way it would have been evaluated had it been part of a property. The result is then written to the result file. For the properties there are two basic categories, those with and those without unbounded variables.

Each P function node and statistical function node in the AST has a symbol table. This table contains name and current value for variables, i.e. the bounded instance, all probes, all unbounded instances and the unbounded variable, used in the condition of that function. The table also contains an invalid

value flag that is set when, for some reason, no valid value for some sub expression of the condition can be found using the current value on the variables.

4.3.1 P on task

For properties without unbounded variables the property is a relational operator applied to two probabilities. If this relation is true then the result of the query is true, if not then the result is false. A probability is either a constant or a P function. The actual work in these queries is to evaluate the P function. The working set of the P is either a task with an instance operator or all tasks, "*". Depending on which of them it is the evaluation is done quite differently. In either case the first thing to do is to push the symbol table of the P on a global symbol table stack.

If the working set of the P is a task then the first step is a loop that evaluates the condition for each of the instances of that set, i.e. all values for the bounded instance variable, as shown in Figure 4.2. For each of those instances the first step is a recursive loop function. This loop function will call itself once for each unbounded instance variable. This gives us a set of nested loops, one for each instance variable. For each iteration of these loops the value of that instance variable in the symbol table is updated. The innermost loop calls the evaluation function for the expression. These nested loops allow evaluation of the condition for every possible combination of unbounded instance variables. But since it only needs to be true for one combination all possibilities will commonly not have to be tested.

Once a combination that makes the condition true is found the loop function aborts and returns *true* to the bounded instance variable loop. If all combinations was tried but the condition was not true for any of them then the loop function returns *false*. Apart from true and false the loop function could also return *invalid*. The reason for an invalid is because the condition could not be evaluated. Most commonly this is because of the cardinality problems when comparing instances. It could also be when using *following* and an instance does not follow any instance in the other set. The bounded instance loop counts how many true and how many invalid results it get. Once all the instances has been iterated the result for the P function is calculated as the number of true divided by how many that did not give an invalid result.

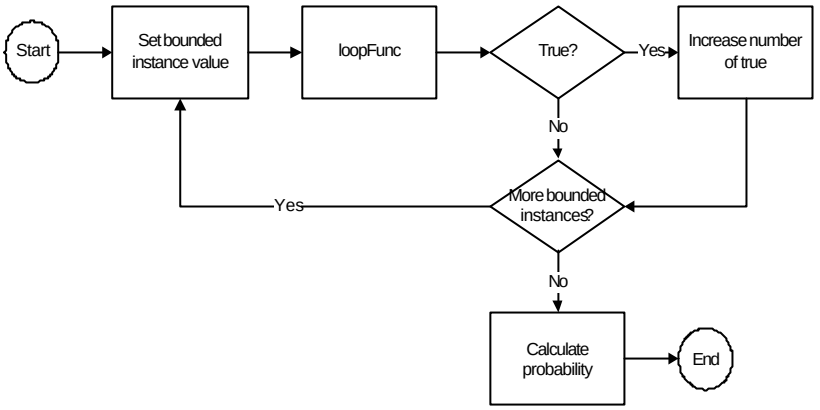

Figure 4.2 Evaluation of P

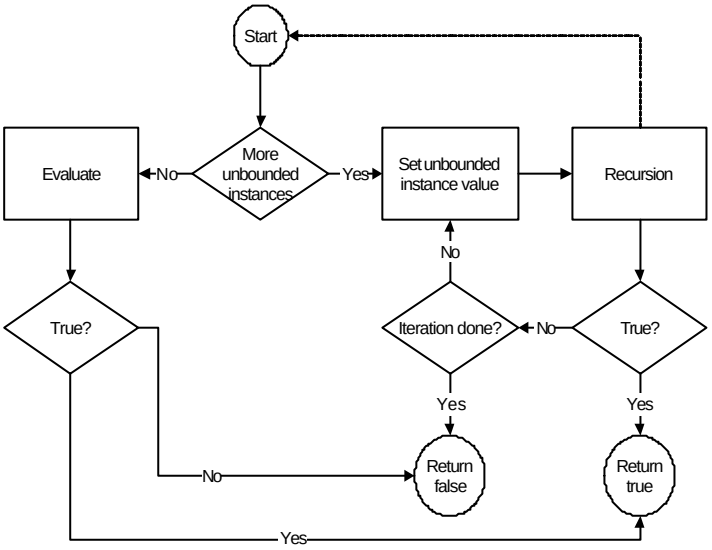

Figure 4.3 Loop function for P on task

4.3.2 P on probes

For P with the working set "*" the basic idea is the same as for P with a task as working set. It is however done quite differently. As these P work on probes over time there are no instances, instead it is time units that is iterated in its version of the loop function. First the point in time when all probes in the expression have a value is found. It is unknown what value, if any, a probe has until the first probe event in the trace for that probe. It is not possible to evaluate an expression containing a probe with no value. Hence, similar to with the instance cardinality problem, the time before all involved probes have values must be excluded. Once that is done the value of the probes is set in the symbol table. The expression is then evaluated. If it is true then the amount of time it was true is stored. This time is from the time of the event when the values were set up to the next probe event for any of the involved probes. For the last values this is up to the end of the log. This is then repeated for all probe events. The value in the symbol table is updated and the expression evaluated. The time for all values that made the expression true is summed up to get the total amount of time units that the condition is fulfilled. The result from the P is then calculated by dividing this total true time with the total time. The total time start at the first point where all the probes had values and ranges until the end of the log.

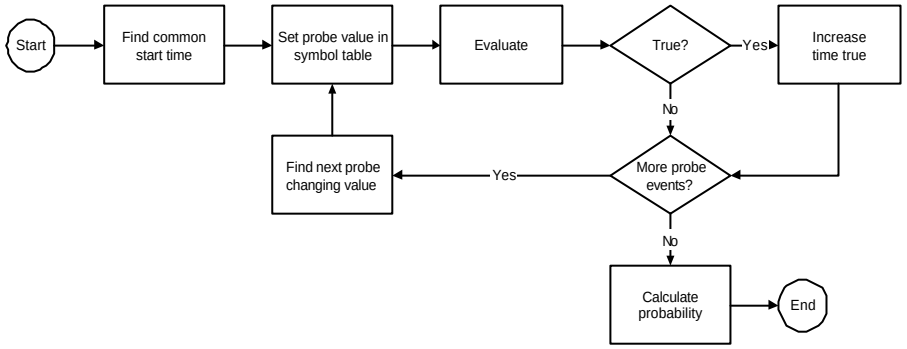

Figure 4.4 Loop function for P on probes

4.3.3 Evaluating

The evaluation of the expression is done the same way for both P on tasks and P on probes. A few recursive functions traverse the AST for the expression evaluating it bottom up. Operator nodes are evaluated by evaluating the operand expression(s) and then applying the operator to the results.

The statistical functions are independent of the expression they are part of. They have their own symbol table, i.e. the instance variable *i* used in the arguments to the statistical function is not the same as the instance variable *i* used outside it like for example in Example 4.1. Thus the result from one of the statistical functions will be the same for all of the bounded instances. To avoid wasting time evaluating the function several times the result of the function is stored once it has been

evaluated. When the function node is reached for the rest of the instances the stored value is simply returned.

$$P(t(i), t(i).resp > avg(t(i).resp)) > 0.9$$

Example 4.1 instance variables in statistical functions

A task node contains the name of the task, its instance operator and the data member to be used. When evaluating a task node the first thing to do is to evaluate the instance operator. If it does not contain a *following* function then the current value of the instance variable is retrieved from the symbol table. If it does have a *following* function then the instance of the task that execute closest after the task instance to be followed must be found. Instances are sorted after start times and thus also end time as instance X(i + 1) cannot end before instance X(i). Because of this the right instance for *following* can be found by searching through the instance list from start. The first instance that ends after the end time of the instance to follow is the instance returned from *following*. Since it is known that the instances are sorted like this it is not necessary to search through the entire list every time. If instance X(i) was followed by instance Y(j) then X(i + 1) cannot be followed by an instance before Y(j). To make use of this the *following* function node keeps a *shortcut pointer* to the previously used instance, in this case Y(j). This eliminates a great deal of unnecessary iterations.

Once the value from the instance variable or *following* function has been retrieved any arithmetic's is applied to get the result from the instance operator. The task node now knows what data member from what instance of the task to use. To get the value of this data member the instance must first be found in the instance list of this task. To avoid searching through the entire list each time the task node, like the *following* function node, keep shortcut pointers. This works here also since the instances are looped through in order. The wanted instance can never be positioned before the previous one in the list. The exception is for unbounded instance variables or, as will be shown later, unbounded variables. With for example one unbounded instance that one unbounded instance will be looped through once for each value on the bounded instance. The instances will still be in order but when the bounded instance is increased the value on the unbounded instance, and hence shortcut pointers as well, must be reset. If the data member wanted was *start*, *end*, *resp* or *exec* then the value is found in the instance and returned. If the data member was not one of them then it was a probe.

The probe values unlike the others are not found in the instance. Instead the event list for the probe must be searched through. The value for a probe data member of a task instance is the value that probe had when the instance begun its execution. As the probe events are sorted on their time shortcut pointers can be used for the probes as well. The probe event for the instance X(i + 1) is the same or a later one as for instance X(i). Once the right probe event is found its value is the value for the probe data member of this task instance. In an AST for a P on probes there are no task nodes. Instead it has probe nodes representing the construct *.probeX. As explained in Section 4.3.2 the loop function for P on probes updates the values for the probes in its symbol table. This is now used when a probe node is evaluated by simply taking the value found in the symbol table.

4.3.4 Unbounded variables

There are two different kinds of unbounded variables, outer unbounded variables and inner unbounded variables. Outer unbounded is when the unbounded variable is a probability, i.e. it is outside of a P function. These are relatively simple, the P function is evaluated exactly like it would have been if there where no unbounded variable. An interval based on the result from this P is then assigned to the unbounded variable. Inner unbounded, i.e. an unbounded variable in the conditional expression of a P function, is quite different. The first task is to find a set of all values that need to be tried as bindings. This is done by first finding the relational operation where the unbounded variable is one operand. The values to try as bindings are all possible values the other operand of that relational operation can take. That expression is evaluated for all possible values on instance variables to get a set of binding values. This set might contain duplets, but there is no point in trying the same value several times as the result would always be the same. Hence any duplets is removed from the set. The set is then sorted as the algorithms used require that. There are three of these algorithms, which of them is used depend on the two operators involved. These operators are the outer, i.e. the one between probabilities in the property, and the inner, i.e. the one where the unbounded variable is one of the operands. All three work somewhat similarly. They take values from the set of possible bindings and set them as the current value of the unbounded variable in the symbol table. Then the query is evaluated for that value using the same functions used for evaluating properties without unbounded variable. The unbounded variable has its own node. When such a node is found the variables current value is taken from the symbol table. The result from the query, *true* or *false* is stored with the value.

If neither operator is strict equal, '=', then the fastest of the algorithms, the quick way, can be used. This one is a divide and conquer style algorithm. It begins with evaluating the expression for the first and last values in the set of bindings. The key in this algorithm is the fact that there will be, at most, one threshold value, i.e. there will be no more than one binding that changes the result. All values after this threshold value will give the one result and all values before will give the other result. Thus it is also known that if the result from the first and the last value is the same then there is no such threshold, i.e. it is either true or false for all bindings. If the first and last is different then the threshold is searched for. This is done by evaluating for the value in the middle of the first and last. If the result from the middle value is the same as the first then the middle is considered the new first. If it is the same as the last then it is considered the new last. This is then repeated until first and last are next to each other. First and last will then give different results and the threshold has been found. The major benefit of this algorithm is that the result can be found after evaluating only a fraction of all the possible bindings. The efficiency of this algorithm is O(log n). The only exception is when there is no threshold. In that case it is O(1).

If the outer operator is strict equal, and the inner any but strict equal, then the middle way can be used. With an outer strict equal there are two thresholds, which is why the fast way can not be used. The first threshold change the result and that result then continues until the second threshold is reached. What this algorithm does is to begin with the first value and then evaluate for each value until it finds the first threshold. The second threshold is later found by continuing the search. When the second is found there is no need to search anymore. Thus every possible value does not need to be evaluated. Still, in most cases, far more than for the quick way need to be evaluated. This method can vary

noticeably in performance. In the best case, the thresholds being the first values, only two evaluations are needed. In the worst case, one of the thresholds being the last value, every value needs to be evaluated. The efficiency of this algorithm is in the best case O(1) and at worst O(n).

If the inner operator is strict equal then a slow way need to be used. With an inner strict equal the whole threshold system falls apart. Theoretically every value could be a threshold. The interval the unbounded variable is bounded to in the end is not really an interval but a set of values or a set of intervals. This means that the query must always be evaluated for every possible binding. The efficiency of this algorithm is always O(n).

Once the thresholds have been found, or in the case of the slow way a full set of results, the interval to assign to the unbounded variable should be constructed. If a threshold was found using the fast way then assigning is a simple task of applying the semantic rule from Section 3. Depending on the operators an interval from -8 to the threshold or from the threshold to 8 is constructed. If no threshold was found two more evaluations are needed. It might be that the threshold is just outside our set of bindings. The query is evaluated with the unbounded variable given the first value – 1 and the last value + 1. If a threshold is found on one of them then an interval is assigned accordingly. If not then the interval will be empty if the results are false or from -8 to 8 if the results are true. For outer strict equal intervals are assigned similarly. The difference is that the interval does not start or end with -8 or 8 by default. Instead the interval is between the two thresholds. If the thresholds are on the first or last value the values just outside the set must be evaluated here also to see if that really is the threshold or if the interval should start/end with infinity. With an inner strict equal assigning is a bit different. If the outer operator is less than, less than equal or the special case strict equal to 0 then intervals should be constructed. As explained in Semantic rule 3.29, the interval (-8..8) is split into several intervals excluding the values that gave the result false. This is done by starting with -8 then finding the first value that gave the result false. The current interval is ended and a new one started with that value. This is repeated until all values with the result false have been added. Finally the last interval is closed with 8. If the outer operator is any other, greater than, greater than equal or strict equal to anything but 0, then a set rather than an interval should be assigned to the unbounded variable. This is done by going through the set of results and adding all values that made it true to the assigned set.

4.4 Performance issues

All times measured and presented here was from tests on a system with a 600MHz processor and 384MB memory running Windows 2000.

Usually most queries would be solved more or less instantly. For these queries reading and compiling the trace take most of the time. A large trace with 300000 events takes about five seconds to read and another two seconds to compile. There is however certain constructs and combinations that are more time consuming to evaluate, in general this concern unbounded variables and instances. As explained in Section 4.3.4 an inner unbounded variable is evaluated in two steps. First all possible values the unbounded can be compared to are found. Then the query is evaluated with the unbounded variable set to each of those values. In that section the three different algorithms used for this was also

explained. The fastest algorithm was used when neither of the two operators are strict equal. This algorithm reduced the number of those possible values that needed to be tested heavily. Because of this algorithm such constructs do not lead to any significant time issues. For a frequently executing task, about 30000 instances, a query with an unbounded variable like Example 4.2 would take about two seconds while one without unbounded like Example 4.3 would take less than a second.

$$P(T1(i), T1(i).resp < X) > 0.75$$

Example 4.2 a standard inner unbounded variable query

$$P(T1(i), T1(i).resp < 50) > 0.75$$

Example 4.3 a standard relation query

When either operator is strict equal the query becomes much more time consuming. With strict equal as the inner operator, as in Example 4.4, every one of those possible values would need to be tested. With the same 30000 instance task as above that query would take somewhere in the region of 40 minutes to evaluate. If only the outer operator is strict equal, as in Example 4.5, then the time required is rather unpredictable. At worst this gives the same scenario as for the inner strict equal. At best the time required would be closer to those queries without strict equal. Commonly the time would probably be closer to the worst case than the best case as finding a value that exactly match a probability, like 0.5 in this case, is rare.

$$P(T1(i), T1(i).resp = X) > 0.5$$

Example 4.4 inner unbounded variable with inner strict equal

$$P(T1(i), T1(i).resp > X) = 0.5$$

Example 4.5 inner unbounded variable with outer strict equal

Unbounded instance variables are time consuming for similar reasons as the unbounded variables. Like for outer strict equal on the variables the time required to evaluate queries with unbounded instance variables are quite varying. At best the first instance makes the property true and nothing more need to be tested. At worst every possibility must be tested. Assuming that the queries written with unbounded instance variables are used to confirm assumptions rather than wild guesses it is likely that most evaluations come closer to the best case than the worst case. For example unbounded instance variables could typically be used to check pre-emption. If the property is formulated on tasks that is known, or at least believed, to pre-empt each other then there will likely be some instances that are true.

As these queries are evaluated by testing every possible combination on the instances the time required to evaluate them increase drastically if more unbounded instances are added to them. The time required for a query without unbounded instances is linear, since there is only the one bounded instance. With one unbounded the time is quadratic, bounded instance * unbounded instance$_1$. With two unbounded it becomes cubic, bounded instance * unbounded instance$_1$ * unbounded instance$_2$, etc. When saying quadratic and cubic here it is assumed that all the tasks have the same number of instances. That is commonly not true but the concept still remains. For every added unbounded instance the total number of evaluations needed in the worst case is multiplied with the size of the largest task with that unbounded instance as instance operator.

Unbounded variables and unbounded instances are similar in how much time they require. Adding an unbounded variable to a query means that the number of required evaluations is multiplied with the number of possible bindings for the variable.

The conclusion of this is that to keep the time it takes to evaluate a query reasonable, the number of unbounded instance variables should be kept low. To combine inner unbounded variables and unbounded instance variables is not recommended. For inner unbounded variables the use of strict equal should be avoided. For the 30000 instance task using other operators reduced the factor multiplied to the number of evaluations needed from 30 000 down to 17, $O(n^2)$ to O(n log n). In time the difference is from 40 minutes down to some seconds. Using strict equal is not very useful to start with. It is very rare that there are properties where something matches an exact probability. In most cases where it would be useful the comparison would be to a probability of 0 or 1. A tip for those cases would be to use >= 1 and <= 0 as those would give the same result but use a different algorithm. In general it is worth considering if the strict equal could not be replaced with a different operator.

The main problem with these queries is that very large amounts of evaluations have to be performed. To come up with a different algorithm that does not require as many evaluations is the only way to reduce the time down towards the levels of queries without unbounded variables. No suggestion for how this could be done has been brought forth. However, two other thoughts, regarding data structures and further shortcut pointers, have been discussed under future development (Section 6.2).

4.5 Test: SQL v PPL

As we where to begin the work of defining the PPL language some tests were performed comparing PPL with SQL. We felt it would be interesting to see how much of the PPL language could be done with a query language like SQL. In addition it could prove useful to have shortly studied a different language when we defining PPL and implementing the tool.

4.5.1 SQL introduction

SQL, Structured Query Language, [10] is a query language for modifying and retrieving information from relational databases. Such databases are based on tables with columns and rows. Each row represents a post in that table. The columns are named and represent the data fields of the posts. Using the SELECT statement we can retrieve columns from tables specified with the FROM statement. Using the WHERE statement we specify conditions that the rows of the selected columns should

fulfil. In our tests we make use of a function Count. It is used in the SELECT statement to count the number of rows in the selected columns.

4.5.2 The tests

We choose four simple queries as we believe SQL will have trouble solving more complex ones. Complexity in a PPL query would come from comparing instances with other instances or by adding unbounded variables. Just extracting instances based on a comparison with a constant should be no problem. We expect SQL to fail however when we need to compare instances with each other or when we compare relative instances.

For the tests we created a database containing one table, *Log*, which is to represent a log file. This table has five columns, *Task*, *Instance*, *Start*, *Resp*, *and Exec*. Each row in the table is an instance of a task. The task-column shows what task and the instance-column shows what instance of that task. Start is the start-time, Resp the response-time and Exec the execution-time of the instance. We had to add the instance numbers as a column in the table in order to be able to compare instances using SQL.

Here we present our four tests. First we present the problem and how it would be formulated using PPL. Then the SQL solution is presented and commented.

A) Count occurrences of values

What is the probability of an instance of a task t having a response time greater than 2?

$$P(t(i), t(i).response > 2) = X$$

Test query 4.1 count occurrences with PPL

With each row of a database table being an instance of a task counting is quite easily done with SQL. The number of rows where the task name is t and the response time is greater than 2 is counted. To get a frequency seems to be more difficult using SQL.

```
SELECT Count(*)
FROM Log
WHERE Log.Resp > 2
AND Log.Task = “t”;
```

Test query 4.2 count occurrences with SQL

B) Count occurrences of sequences

What is the probability of two consecutive instances of a task t both having a response time greater than 2?

$$P(t(i), t(i).response > 2 \text{ AND } t(i+1).response > 2) = X$$

Test query 4.3 count occurrences of sequences with PPL

Similar to A) this can be done by counting rows. First two copies of the table are made using the AS-command in order to compare rows. Then all rows are found where both task fields are t, the difference between the instances is 1 and the response times are greater than 2. The last line "AND L1.Instance<L2.Instance" is used to filter out doubles. The filtering could also, more efficiently, have been done by removing the abs() function from a previous line.

```
SELECT Count(*)
FROM Log AS L1, Log AS L2
WHERE L1.Task = "t"
AND L2.Task = "t"
AND abs(L1.Instance-L2.Instance)=1
AND L1.Resp>2
AND L2.Resp>2
AND L1.Instance<L2.Instance;
```

Test query 4.4 count occurrences of sequences with SQL

C) Pre-emption.

What is the probability of a task t1 being pre-empted by a task t2?

$$P(t1(i), t1(i).start < t2(j).start \text{ AND } t2(j).start < t1(i).end) = X$$

Test query 4.5 pre-emption with PPL

Two copies of the Log table are made in order to be able to compare rows. Then all combinations of rows are found where the first task is t1 and the second task is t2 and where the start time of t1 is greater than the start time and less than the end time of t2. Those rows can then be counted.

```
SELECT Count(*)
FROM Log AS L1, Log AS L2
WHERE L1.Task = "t1"
AND L2.Task = "t2"
AND L1.Start > L2.Start
AND L1.Start < (L2.Start+L2.Resp);
```

Test query 4.6 pre-emption with SQL

D) Separation

Is the probability of instances of two tasks, starting within 3 time units, having a response time greater than 2, 0.9.

P(t1(i), t1(i).resp > 2 AND t2(j).resp > 2 AND abs(t1(i).start – t2(j).start) < 3) = 0.9

Test query 4.7 separation with PPL

This also partially possible using SQL, like before the first step is copying the table. Then all combinations of rows are found where the first task is t1 and the other is t2, both response times are greater than 2 and the difference between the start times are less than 3. As in all previous tests only the number of occurrences is found. Here one more problem, if so related to the first one, was encountered. No method of comparing the result with 0.9 as asked in the PPL query could be found.

```
SELECT Count(*)
FROM Log AS L1, Log AS L2
WHERE L1.Task = "t1"
AND L2.Task = "t2"
AND L1.Resp > 2
AND L2.Resp > 2
AND abs(L1.Start-L2.Start) < 3;
```

Test query 4.8 separation with SQL

4.5.3 Summary

It seems we have underestimated SQL. When it comes to extracting sets of instances it suffices. One problem was when comparing sequences of instances. A query like t(i).exec < t(i+1).exec could not easily be asked in SQL as it has no way of asking for "the next row" or "two rows down". Being intended for relational databases SQL has no need for such queries, the choice of adding a column for instances, at least in these test, solved that problem.

The problem of getting probabilities might be possible to solve. Integrating the query in some other environment would most definitely make it possible as it is merely the task of dividing the result from two SELECT COUNT queries.

From what we learned from these tests it is possible that SQL could be used as a back end for a PPL evaluation tool.

4.5.4 Follow-up

When defining PPL we made some changes that affect the test made in our comparison between PPL and SQL. Most noticeably we added the working set to the query in order to avoid ambiguity, see Section 3 for details. This would be handled in the SQL queries by specifying the copy of the log table that represent the set rather than * in the SELECT statement.

More problematic is the evaluation of probes over time. The table for probes would have columns for start time, data and how long that value was held. A query containing a single probe could be solved. Those would be solved similar to a task query. The events, i.e. rows, where the condition was fulfilled would be extracted using SQL. The probability would then easily be calculated from the time held row. If the query contains several probes, like Example 4.6, this can not always be done. The two probes do not change value at the same time. For example the values could be such that the condition is false at first. But then probe20 change value to one where it is true. After a while probe30 also change value to one such that the condition is false again. In this case we would need to extract a part of the time held field of one of the rows. This cannot, at least not easily, be done.

$$P(*, *.probe20 > *.probe30) = X$$

Example 4.6 a probe query

Furthermore the task of creating a translator from PPL to SQL is not always easy. A PPL construct can not always be simply changed to a counterpart in SQL. This is most noticeable for queries with several instance operators and especially for relative instances. Although not impossible it would be a quite complex task. The gain of simplifying our PPL tool with a SQL backend would be lost in the work on translating the queries.

In conclusion we decided that using SQL is not suitable.

According to our tests using SQL would not be faster than our non SQL implementation. The first two test cases, A and B, are solved instantly both by SQL and PPL. Other more heavy queries take more time for both. PPL was faster solving test C while SQL solved test D faster. Both test C and D was run several times with slight modifications in the values to get worst, average and best cases in terms of how many evaluations would be needed by PPL. The only noticeable difference between the two languages was when test D was changed such that the response time of t2 should be less than 0 instead of greater than 2. This causes a worst case for PPL since it is never true. SQL however managed to solve this instantly, probably by somehow seeing that this would never be true. In all

other cases the difference was small and overall neither could be considered faster. When looking at this result we should however take into consideration that our SQL implementation is probably not optimal.

5 Testing the tool

Testing was done in order to verify the functionality of the tool. The approach used was to first manually calculate the query and then comparing that result with the result from the tool. See Appendix D for some examples of test calculations. The results for the test queries were stored so that the tests could be redone every time bug fixes or other changes was made, i.e. regression testing. If the test queries, the trace and the results for those queries for that trace are kept then this method could also be used in the future.

A small trace was created specially for this. This trace contains four tasks, Task_ZERO, Task_ONE, Task_TWO and Task_FF. The tasks were given the ids 0, 1, 2 and 255. A task id is represented by an unsigned char making 255 the highest value. Normally tasks have a low id so here we gave one the highest possible to make sure it did not cause any unsuspected problems. The tasks were given various properties such that various special cases would occur. In addition they were kept small, between 6 and 16 instances, so that it would be possible to perform the queries by hand. The trace also contains six probes with different number of events. Two of them have only one event. One of those two is also the very last event of the trace. Like the tasks the probes was made with a number of possible problems in mind.

First the basic unary and binary operators needed to be tested. This was done by running simple queries applying them on a task and a constant. These kinds of queries are quite trivial and thus a relatively small number of examples were considered enough to verify their correctness.

The first major part to be tested was instances and the various cardinality problems involved with them. The most basic version of that problem is when arithmetic's is applied in the instance operator. This was tested by a number of queries applying binary operators on a task instance and the same task instance with a number added or subtracted in the instance operator. With minus the *n* first instances was excluded and with plus the last *n* instances was excluded. This was checked by debugging the tool and making sure that the result from evaluating the condition of the P function was INVALID for the instances that should be excluded.

The next version of the cardinality problem is when comparing the same instance of different tasks with different rates. This likewise was tested by checking that the result from the P was INVALID for those instances that should be excluded.

The cardinality problem involving AND and OR is a bit special as they are not applied directly to tasks. They instead need to handle invalid values in general. The most common reason for these invalid values are however the cardinalities. AND is done the same way as the other binary operators and thus required no extra testing. For OR however it needed to be tested that it would give the result INVALID if both its operands are invalid and not if only one is. This was tested by looking at OR nodes and make sure they returned the expected result for any combination of operands.

The *following* function was tested for two things. First to make sure it could properly handle the case were some instances do not follow any instance. The result for those instances should be INVALID. Like the cardinality problem this was tested by watching the result for each instance of P functions. This could specially be tested using Task_ONE as the first instance in the trace belongs to that task. Hence there is no instance of any task that its first instance follows. The second issue is to test that the function returns the correct instance. The expected result was calculated manually from the end times of the involved tasks. These results were then matched with the results given from the function.

The operators are no different if they are applied to tasks or probes. Thus there is no need to test them especially for probes. What needed to be tested for probes is their loop function. That it sets values and calculates time correctly and that the common start time of the probes in a query is found correctly. Some special cases that needed to be looked at were a probe that never changed value and a probe whose only event is the last event of the log. The last event ends the log, thus that probe will set its only value on the end time. This could have cause problems as it had a value for 0 time units. This was tested by watching the time true, total time and then the end result calculated by the function. The same values was calculated manually and then compared.

The probes also needed to be tested when used as a data member of a task. Firstly to make sure that the correct probe is used, but more that all cases where a probe did not have a value for some task instances was handled correctly. This is handled by the function that gets all values from data members of tasks. Thus testing this for probes also tested that all the other data members, *start*, *end*, *resp* or *exec*, was retrieved correctly. This was tested by watching the values returned from this function to see that they were the expected ones.

The statistical functions when not using a subset is quite straightforward. The only testing required there is to make sure they give the correct result. Like many other features this was tested by manually calculating the results and comparing with the results returned from the function. For a function with a subset it also needed to be verified that this subset was constructed properly. This was tested by watching the subset passed to the function and comparing it to what the subset was expected to be. One thing learned from this was that it is possible to get empty subsets.

Unbounded instance variables are not noticeably different from the bounded instance variables as far as most of the functionality is concerned. During evaluation all instance variables are treated the same no matter if they are bounded or unbounded. What mainly needed to be tested for these was the loop function. To make sure that it stopped looping when, and only when, a true result is found or every possibility has been tried. And also to make sure that the way invalid and false results were treated did not cause any errors. The handling of invalids had been known to cause trouble. Thus great time was spend on testing numerous queries that would result in invalid values for varying instances. Part of the problem with the invalid values was how they are passed from the source of the invalidity up to the loop function. Thus the evaluation was stepped through node for node for some queries. That is however quite time consuming and not something that could be done for a large quantity of queries. Hence most of the testing here had to be done by watching the result from the loop function or results from the evaluation to the loop function.

Unbounded variables have been the most error prone feature of the tool. To find the values that are valid bindings are relatively simple. As explained in Section 4.3, this process uses the same functions as when evaluating queries without unbounded variables. The complexity lies in the assigning process for inner unbounded variables. Assigning an outer unbounded is not complex as it only assign one value depending on one operator. For an inner unbounded there can be several values. They also depend on a combination of two operators. Thus there are far more possible ways of assigning. Furthermore the amount of combinations is quadrupled because of the ways the query can be constructed. The meaning of the outer operator varies depending on if the P with the unbounded is the left or right operand of the property. The same way the meaning of the inner operator vary depending on if the unbounded variable is its left or right operand. Apart from all these possible combinations there is also numerous special cases that need to be considered. For example if all values are true or false then values outside the set of values must be tried for some combinations. Because of all this the inner unbounded variables was the most extensively tested feature. Testing this was done by calculating results and constructing bindings by hand. These intervals were then compared to the ones constructed by the tool.

As mentioned this test trace was quite small in order to make it possible to manually calculate the results for queries. Such a small trace is however not realistic. Thus to test the speed and robustness of the tool some tests on larger traces was also needed. During these tests queries was run on tasks with up to 30000 instances. The results of these tests showed some serious time issues. Some of what could be considered standard queries would take several minutes or even hours to evaluate. This led to several optimisations, most noticeably the quick version of the unbounded evaluation algorithm, see Section 4.3.4, and the shortcut pointers when accessing data members of task instances, see Section 4.3.3.

6 Development of PPL

In this section we present the development of PPL from where we started to where it is today. We present how we first planned it and what changes was made and why they where made. Our development of PPL has been an iterative process. Changes have been made up to quite late stages. The latest of the changes however not as noticeable as the earlier as they where merely tweaks or minor features. We also discuss future development. Looking at what could have been done differently as well as features that could be added on.

6.1 Current development

As mentioned PPL had previously been outlined [13][14]. Naturally our work was based on this. This outline was quite restrictive in what was allowed. Originally we followed in that path. For probes, or message queue as they where considered at this point, we allowed very little. They could only be used in a comparison with a constant or unbounded variable. No logical or arithmetic operations were allowed. Unbounded variables were restricted similarly. They could be compared to a task or a probe. No arithmetic's could be applied to them and they were not allowed to be part of logical operations. Furthermore they could not be combined with unbounded instances. When it came to the structure of the queries we went the other direction making it less restrictive. The originally suggested grammar was quite inflexible. For example the P function was defined as always being the left operand of the property and probes was defined as always being the left operand in the comparison. The P could also only be compared to a constant or unbounded variable. There where no reason for this so we made our grammar more flexible in these regards. We even went one step further as well allowing logical operations between P functions. That feature however was soon scrapped as it was deemed useless. Some small functions like *abs*, *min*, *max* and *avg* was added. To finalise the basis of our grammar we extended the arithmetic's allowed. Originally only the operators '+' and '-' was allowed. Those could be applied to a task and a constant or two tasks. We introduced '/', '*' and unary minus and allowed any arithmetic expression that could be constructed using those five operators, parenthesises, constants and tasks.

With that the first version of our grammar was complete. When the work of defining the semantics for this grammar begun we soon realised some changes was needed. The first problem was encountered already when defining the P function. We could get ambiguous queries! The solution was the simple but effective choice of explicitly stating what task the query should be asked on. With that the working set was added to P.

The unbounded variables were the next feature to cause changes. An outer unbounded was always clear, simply evaluate the expression it was compared to and assign the result. The inner unbounded variables needed a bit more work. Strict equal had been foreseen to be more troublesome and was thus saved for later. Starting with the other operators it was soon clear that the two suggested semantic rules were not enough. What we found out was that not only the outer relation decided the result but the combination of the outer and inner. This doubled the amount of rules up to four. The problem with using strict equal was that we could get undecidable queries. When considering this we came to the conclusion that we did not need to restrict the use of strict equal as had been suggested. It would not

be difficult to attempt to evaluate a possibly undecidable query. If the query turned out to be undecidable we could give an answer saying that. This was the first of what would turn out to be several times we relaxed the restrictions on the unbounded variables. The second one was soon to follow. Looking at the semantic rules it was clear that allowing logical operators in queries would not cause any trouble. The process of binding an unbounded variable would not at all be affected by allowing this.

Comparing instances of a task with the same instance of a different task is not very useful if the tasks have different rates. This was not new to us. But we had not considered allowing more useful comparison between different tasks until such a feature was requested. We came up with two possibilities. The first was to redefine the instance operator such that comparing the same instance of two tasks would be to compare instances that were close in time. The second was to use some function to perform this mapping. The latter option was chosen and resulted in the *following* function. It was at this time that we also came up with the idea of adding the sequencing feature to the instance operator.

At this point the probes were quite different than they turned out to be in the end. As mentioned they where not considered probes but message queues. On these queues we could ask queries about their size. Before we got as far as considering the semantics for these queues they had already turned into probes. These probes were tied to task instances. The observations from the probe would be written to the currently executing instance. In a query the probe value would be accessed as a data member of a task instance. A problem soon arose in the fact that queues, or other probes for that matter, can be shared among tasks. How do we say that a probe should never be 0 if the observations from that probe are spread over instances of several tasks? The solution was the feature of using the union of all tasks, '*'. For this construct the old message queue restrictions lived on, i.e. no arithmetic's and no logical operators. Not too long after yet another question was raised about the probes. How useful was it to look only at the values from the observations? Would it not be better to look at time? The probability of a probe having a certain value would be evaluated quite strangely if looking only at the observed values. For example if a message queue is given the size 0 on half of the observations does not mean it has size 0 half the time. It might perhaps be that the size is only 0 a short time between two other values that are kept for longer periods of time. Thus the probabilities we would calculate using observations would be wrong. The probes were once again changed, now to represent the value over time.

A final addition before implementation begun was to allow the small statistical functions, *min*, *max*, *avg* and *median* to be used as stand alone queries and not only as part of a P.

As the tool was implemented one thing eventually became clear. Many of our restrictions were not needed. In fact for several of them it would be more work to restrict them than to allow it. This was especially true for the probes. More or less all restrictions on them were removed so that they could be used in expressions just like task instances in general. The same was true for unbounded instances. We now allowed combining them with unbounded variables. A main problem with combining unbounded instances and variables had not been how to solve them but how to solve them efficiently.

In most cases this is still not possible. However restrictions would only remove the possibility not make it more efficient.

The assign process when evaluating unbounded variables provided plenty of work. It turned out that the new semantic rules we had added for this previously was not enough. We needed to create one rule for every combination on the inner and outer operator. Luckily it turned out that some of the combinations resulted in the same rules. However not in the general fashion that we had first said. Furthermore our decision of allowing flexibility in the structure of the query came back to haunt us. Since the unbounded variable could be both left and right operand of the inner operator and the P both left and right operand of the outer operator the number of possibilities was quadrupled.

The last implemented change, except for what seemed like a never ending stream of tweaks to the unbounded variables, was some changes to the statistical functions. Instead of applying the function to an entire set it was suggested that a subset could be first be chosen and the function then applied to this subset. This was done by allowing the statistical functions to imitate the P function. Apart from a set, which in the case of the statistical includes a data member as well as a task, a condition should be given as an argument. The function would then be applied only to the subset that fulfilled the condition. Spawned from this was the idea of a subset function. We already had the functionality to create a subset. Why not allow this to be used directly?! It could be interesting to be able to extract a subset of, or all, response times or some other value. This resulted in the subset function that prints a subset to a file.

6.2 Future development

Despite, or perhaps just because of, being the most changed feature during our development of PPL, the probes could be done differently. We say in the semantics, Section 3.4, that we cannot reason about instances on probes. Initially that statement was true but at this point it is somewhat questionable. The probes could be done like instances. If we considered everything events, then we would have task events and probe events. The same way task events are compiled into task instances, probe events could be compiled into probe instances. From there we could remove the use of the construct *.probeX to represent all observations of a probe over time. We would also remove the probe data members from the task instances. From this we get two types of events that could be treated equal. The probe instance would have start and end time just like the task instances. It would have its value and some equivalents of execution time representing how long that value was kept, i.e. its end time minus its start time. These probe instances would be used exactly like task instances with the instance operator etc. The only difference would be the slightly different data members in the two types. In a P or statistical function we would still get differences depending on the type of event in the working set. If it is a probe event we would get the probability over time like we get now using '*' as the working set. This would allow more powerful queries to be formulated. An example that has come up is to see if probes change value during the execution of some task. Currently such a query is impossible to formulate. As we have defined it the probe data member of a task instance represent the value that probe had at the start of that instance. In general it would be much more straightforward to formulate a query on a relation between a task and a probe. The drawback is that this also opens up several possibilities to formulate strange queries. For example comparing start times of a probe with

start times of a task. Such possibilities already exist in a lesser degree. It would not cause any trouble and is not necessarily something that needs to be taken care of. The only effect would be that the answer from such a strange and pointless query might likely be equally strange and pointless.

One of the few remaining restrictions on unbounded variables is that the query may only contain one of them. With the restriction that we may not compare two unbounded variables with each other it would not be impossible to allow any amount of unbounded variables. To evaluate this would be no different than evaluating a query with several unbounded instances. The problem lies in assigning them. The assign process for inner unbounded variables is already the most complex part of the language. To redefine the semantic rules to handle tuples etc would not be impossible but require a great deal of effort. Another problem would be the time it would require to evaluate such a query. To combine unbounded instances and unbounded variables was initially restricted because we can not do it in an efficient way. To evaluate a query containing a few unbounded variables and tasks with a decent amount of instances would possibly take several hours.

As the time issues when evaluating queries containing unbounded instance variables are directly related to the large amounts of evaluations needed a small improvement in the evaluation function can make a big difference. Changing data types from double to long wherever possible made a big difference for time consuming queries. With that in mind it could be possible to further improve performance by optimizing the various data structures and algorithms used.

To speed up the evaluation time when using unbounded instance variables the shortcut pointer concept could be applied to the loop function in the tool. The instances are sorted on *start* and *end* times. If we remembered the last instance that made a property true we would have to search less for those two data members. For the other data members, *exec*, *resp* and probes, there would be no difference. As those members are not sorted it does not matter at what instance we begin the search. We would however have to start over with the first instance if we did not find a valid one before the last instance. This would not affect the worst case then as we still need to try for every possibility for those unsorted members. For queries using the *start* and *end* members the average time required to evaluate the query would be lowered.

Of the two possible additional features suggested in the problem description, Section 1.2, neither was implemented. The first feature was to add macros to allow more complex queries to be written more easily. Before being evaluated any macros in a query would be translated to pure PPL. Hence this feature could quite easily be added onto the existing tool as it would not require any changes. The translation of the macros is merely one additional step to be taken before parsing the query. This could be done in the PPL evaluation tool but also in another application that uses it. In that case the macro would be translated before running the tool at all.

The other feature was to use several sets of data. We have considered two versions of this. The simplest one would be to evaluate the query repeatedly on different traces and then give one answer from each trace. These answers could possibly be merged to give some average answer. This could be done outside the tool by simple running it for the same query on various traces. The other way to do

simultaneously work on several traces would be to combine the traces before evaluating queries on them. For this we have two ideas. The first would be to connect all the traces making one long trace out of them. With this one would need to deal with that the various times, start time etc, would reset at the connection points. A straightforward solution to that would be to add the total time from all previous logs to each start and end times. That way the times would never reset and there would not be any noticeable difference between the merged log and a single larger log. Another possibility would be to merge the traces into some average of them. From all the traces we create one trace that has its properties derived from the traces we merge. For this to be useful it would have to be assumed that the various traces were somewhat similar. That is however probably the case for all versions working with several sets of data. Only in this case it is a necessity to be able to even perform the merger.

7 Conclusions

In this work we have defined the PPL language. We have explained the syntax of all the operators and defined their exact semantics. From the previously suggested version of PPL [14] we have extended the arithmetic operations allowed. Statistical functions have been added to be used both inside a property as well as on their own. The data model has been defined with the four time members and probe members. For probes the use of '*' as working set has been introduced to look at the values over time rather than at the start of task instances. As discussed in Section 6.2 those probes could have been done differently, treating them as instances, which would make it possible to formulate more properties than now. For unbounded variables we have defined the semantics explaining how they are bounded to intervals depending on the operators used in the query.

A PPL tool has been implemented that given a batch of PPL queries produces a file with results. This tool is fully able to evaluate any query allowed by our definition of the PPL language. The tool is robust and in most cases fast. The special constructions that can not be solved fast have been discussed in Section 4.4. In that section we explain the reason for the issues as well as how they can be avoided. We had two suggestions for additional features in the tool, support for macros and to be able to simultaneously work on several execution traces. Neither was implemented and instead left as possible future developments as shortly discussed in Section 6.2.

References

[1] R. Agrawal and R. Srikant, Mining sequential patterns, Proceedings of the Eleventh International Conference on Data Engineering, 6-10 March 1995, Pages:3 - 14

[2] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley Pub Co, ISBN: 0201100886

[3] J.H. Andrews and Y. Zhang, General Test Result Checking with Log File Analysis, IEEE Transactions on Software Engineering, v. 29, no. 7, July 2003, pp. 634-648.

[4] James H. Andrews and Yingjun Zhang, Broad-Spectrum Studies of Log File Analysis Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000), Limerick, Ireland, June 2000, pp. 105-114.

[5] James H. Andrews, Testing using Log File Analysis: Tools, Methods and Issues. Procs. 13th Annual International Conference on Automated Software Engineering (ASE'98), Honolulu, Hawaii, October 1998, pp. 157-166.

[6] I. Bratko and D. Šuc, Qualitative data mining and its applications,. Proceedings of the 25th International Conference on Information Technology Interfaces 2003, ITI 2003., June 16-19 2003, Pages:3 - 8

[7] Charles Donnelly and Richard Stallman, Bison The YACC-compatible Parser Generator, November 1995, Bison Version 1.25
http://www.cs.princeton.edu/~appel/modern/c/software/bison/bison_toc.html

[8] M.S. Feather, Rapid Application of Lightweight Formal Methods for Consistency Analyses; IEEE Transactions on Software Engineering, November 1998, Vol 24 No 11: 949-959,

[9] J. Han, Y. Fu,W.Wang, K. Koperski, and O. Zaiane. DMQL: A data mining query language for relational databases. In Proceedings of the ACM SIGMOD International Conference on Management of Data, 1996.

[10] J. Melton, SQL language summary, ACM Computing Surveys (CSUR) Volume 28 , Issue 1 March 1996, Pages: 141 – 143, ISSN:0360-0300

[11] Vern Paxson, Flex, version 2.5, A fast scanner generator, Edition 2.5, March 1995
http://www.cs.princeton.edu/~appel/modern/c/software/flex/flex_toc.html

[12] S. Qiao and H. Zhang, An Automatic Logfile Analyser for Parallel Programs, Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Vol. III, Editor: H.R. Arabnia, Las Vegas, Nevada, USA, June 28 - July 1 1999, pp. 1371-1376

[13] A. Wall, J. Andersson, C. Norström, Probabilistic simulation-based analysis of complex real-time systems . Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 14-16 May 2003, Pages:257 – 266

[14] A. Wall, Architectural Modelling and Analysis of Complex Real-Time Systems, PhD Thesis, Mälardalen University Press, September 2003. Pages 125-154.

Appendix A. The grammar of PPL

in BNF (Backus Naur Form)

```
<query>       ::=  <property> ";" <query>
                |  <property>

<property>    ::=  <value> <relop> <value>
                |  <function>
                |  subset "(" <arg> ")" ">" FILENAME

<value>       ::=  "P" "(" ID "(" ID ")" "," <cond> ")"
                |  "P" "(" "*" "," <cond> ")"
                |  PROB
                |  <unbounded>

<cond>        ::=  <expr> <moreexpr>
                |  <expr>

<moreexpr>    ::=  <logop> <expr> <moreexpr>
                |  <logop> <expr>

<expr>        ::=  <exp> <relop> <exp>
                |  <exp> <relop> <unbounded>
                |  NOT "(" <cond> ")"
                |  "(" <cond> ")"

<exp>         ::=  <term> <moreterms>
                |  <term>

<moreterms>   ::=  + <term> <moreterms>
                |  - <term> <moreterms>
                |  + <term>
                |  - <term>

<term>        ::=  <factor> <morefactors>
                |  <factor>

<morefactors> ::=  * <factor> <morefactors>
                |  / <factor> <morefactors>
                |  * <factor>
                |  / <factor>
```

```
<factor>      ::=  "(" <exp> ")"
               |   abs "(" <exp> ")"
               |   <function>
               |   CONST
               |   <task>
               |   "*" "." probe NUM
               |   - <factor>

<function>    ::=  min "(" <arg> ")"
               |   max "(" <arg> ")"
               |   avg "(" <arg> ")"
               |   median "(" <arg> ")"

<arg>         ::=  ID "." <data member>
               |   ID "(" ID ")" "." <data member>
               |   ID "(" ID ")" "." <data member> "," <expr>
               |   "*" "." probe NUM
               |   "*" "." probe NUM "," <expr>

<unbounded>   ::=  ID

<task>        ::=  ID "(" <instance> ")" "." <data member>
               |   ID "(" <following> ")" "." <data member>

<instance>    ::=  ID
               |   ID + <num>
               |   ID - <num>

<num>         ::=  "[" NUM ".." NUM "]"
               |   "[" - NUM ".." NUM "]"
               |   "[" - NUM ".." - NUM "]"
               |   NUM

<following>   ::=  following "(" ID "(" <instance> ")" ")"
               |   following "(" ID "(" <instance> ")" ")" + <num>
               |   following "(" ID "(" <instance> ")" ")" - <num>

<data member> ::=  start
               |   end
               |   resp
               |   exec
               |   probe NUM
```

```
<relop>       ::=  <
                |    >
                |    <=
                |    >=
                |    =

<logop>       ::=  AND
                |    OR



PROB     ::=  {x: x∈R && 0 <= x <= 1}
CONST    ::=  {x: x∈R}
NUM      ::=  {x: x∈Z}
ID       ::=  LETTER(DIGIT|LETTER|'_')*
FILENAME ::=  '"'ID('.'ID)*'"'
```

Appendix B. The abstract syntax tree nodes

All nodes have the same type, treeNode.

treeNode	
A node in the AST.	
kind	An identifier for what kind of node this is.
unbounded	A flag showing if this node or one of its sub trees contains an unbounded variable.
Node	A union of the various node types. Which one is used here decide what kind of node it is.

Node is a union of the following structures.

tQuery	
The root node of each query in the list of queries from a query file.	
property	The property that make up this query.
pNext	The next query.

tProperty	
The root node of a property. A relation between two probabilities.	
left_operand	The sub tree containing the left operand of this property.
right_operand	The sub tree containing the right operand of this property.
op	The operator in this property.
symbols	The symbol table for the property. Only for any outer unbounded variable.

tP	
The node for a P function on tasks.	
task_name	Name of the task in the set.
task_id	Id of the task in the set.
instance	Name of the instance variable in the set.
expr	The sub tree containing the conditional expression of this P function.
symbols	The symbol table for this P function.

tPlt

The node for a P function on probes.	
expr	The sub tree containing the conditional expression of this P function.
symbols	The symbol table for this P function.

tUnary

The node for a unary operator.	
op	The operator.
expr	The sub tree for the operand.

tBinary

The node for a binary operator.	
left_operand	The sub tree containing the left operand of this operator.
right_operand	The sub tree containing the right operand of this operator.
op	The operator.

tFunction

The node for a statistical function. (plus *abs* and *subset*)	
function_name	Identifier for what function it is.
argument	The sub tree containing the argument to the function.
subset_file_name	The name of the file that the result should be written to if the node is for the function *subset*.
is_calculated	A flag to notify if the result of this function is already calculated.
value	The result of this function. To be used when the is_calculated flag is set.
symbols	The symbol table for this function.

tFunctionSubset

The node for a subset argument to a function, i.e. an argument containing a condition.	
task_name	Name of the task in the working set of this subset.
task_id	Id of the task in the working set of this subset.
instance	Name of the instance variable in the set of this subset.
field	Name of the data member of the instances from the subset that the function should be applied to.
field_id	Id of the data member.
expr	The conditional expression of this subset.
pTask	A shortcut pointer to the task in the working set.
pInstance	A shortcut pointer to the previously used instance of the task in the working set.
pProbe	A shortcut pointer to the previously used probe event when the data member in the set is a probe.

tTask

The node for a task.	
task_name	Name of this task.
task_id	Id of this task.
field_name	Name of the data member to be accessed.
field_id	Id of the data member to be accessed.
instance	The sub tree for the instance operator of this task.
pTask	A shortcut pointer to this task.
pInstance	A shortcut pointer to the last used instance.
pProbe	A shortcut pointer to the last used probe event when the data member is a probe.

tProbe

The node for a Probe.	
probe_name	Name of this probe.
probe_id	Id of this probe.

tInstance

The node for a instance operator. If no arithmetic's is used then op is '+' and num a constant with value 0.	
variable_name	Name of the variable used in this instance operator.
variable_id	Id of the variable used in this instance operator.
op	The arithmetic operator to be applied in this instance operator.
num	A sub tree containing the constant or sequence that op is applied to.

tFollowing

The node for a following function.	
followed_task	Name of the task to be followed.
followed_task_id	Id of the task to be followed.
followed_instance	Instance of that task to be followed.
op	Arithmetic operator to apply to followed_instance.
num	The constant or sequence op should be applied on.
pTask	A shortcut pointer to followed_task.
pInstance	A shortcut pointer to followed_instance.
pProbe	A dummy probe pointer. (Required by the get task member value function but never actually used.)
pFollowingTask	A shortcut pointer to the task that is to follow followed_task.
pFollowingInstance	A shortcut pointer for the instances of the task following.

tFloatConst

The node for a float constant.	
value	The value of this constant.

tIntConst

The node for an integer constant.	
value	The value of this constant.

tUnbounded

The node for an unbounded variable.	
variable_name	The name of the unbounded variable.

tSequence

The node for an instance operator sequence.	
start	The first value of the sequence.
end	The last value of the sequence.

Appendix C. Summary of the code

_ppl.lex.c

Contains the scanner generated from _ppl.lex.

_ppl.yac.tab.c -.h

Contains the main function and the parser generated from _ppl.yac.

ast.c -.h

Contains the structure for the abstract syntax tree nodes, tTree, and a create function for each node kind.

key global variables

tree_root	The root node of the ast.

key functions

tTree copyTree(tTree node)

Creates a copy of a tTree.	
node	The root of the tree to be copied
return	The root of the copy

void freeTree(tTree tree)

Deallocates the memory for a tree by recursing the tree and freeing nodes bottom up.	
node	The root of he tree to be freed

ast_check.c -.h

Contains various functions for validating an abstract syntax tree.

key functions

bool_t typecheck(tTree node)

Check that all expressions in the tree have valid types. The error found is set with setError.	
node	The root of the ast to check.
return	TYPE_ERROR if an error was found

bool_t namecheck(tTree node, char **error)

Checks that all task and probe names in the queries exist in the log. Inserts id of found tasks into nodes and symbol tables	
node	The root of the ast.
error	Is set to the name that is not found. Will not be set if we return TRUE.
return	TRUE if no error was found, FALSE otherwise.

int unboundedcheck(tTree node)

Counts number of unbounded variables in an ast. Marks the sub trees that contain an unbounded variable. Sets an error if we find an unbounded variable in a function.	
node	The root of the ast.
return	Number of unbounded variables found.

ast_translate_sequence.c -.h

Contains various functions used to translate instance sequences into several expressions connected with AND. The function translateSequences is called to start the process. The work is done by repeated calls of the function translateSequence.

key functions

void translateSequences(tTree node)

Translate all sequenced instances to ANDs. Loops calling translateSequence repeatedly until it return that it is done.	
node	The root of the tree in witch sequence nodes are to be translated.

bool_t translateSequence(tTree node)

Translate a sequenced instance to ANDs. Once one sequence is translated it returns.	
node	The root of the tree in which sequence nodes are to be translated.
return	TRUE when not done, FALSE when done.

evaluate.c -.h

Common functions for evaluating queries.

key functions

double evaluate(tTree node, FILE *output_file)

Evaluates a PPL query. This function is used to start evaluation of any query.	
node	The root of the ast for the query to be evaluated.
output_file	The file that the result is to be written to. Used for subset and unbounded variables.
return	If P then TRUE or FALSE, if a function then its result. For unbounded variable queries the returned value is ignored.

void assignOuter(double value, int op, FILE *output_file)

Constructs the interval of valid bindings for an outer unbounded variable and writes it to the result file.	
value	The threshold value in the interval to be assigned.
op	Id of the assign operator.
output_file	The result file.

void assignInner(tTree node, bool_t *unbounded_bool_set, double* unbounded_values_set, long nof_instances, long threshold, FILE *output_file)

Constructs the interval of valid bindings for an inner unbounded variable and writes it to the result file.

node	The P or Plt node that called this function. Used when we need to check for values outside unbounded_values_set.
unbounded_bool_set	A list with the results from evaluating with each value. Is connected to unbounded_values_set such that unbounded_bool_set[1] is the result for the value unbounded_values_set[1].
unbounded_values_set	A list with all the values we tried as bindings.
nof_instances	Length of the lists
threshold	The threshold in the unbounded_bool_set list, i.e. the index where it switch between true and false. -1 if not already found.
output_file	The result file.

double* evalUnboundedCompareExprP(long *nof_instances, tTree expr)

Evaluates the expression an unbounded variable is compared to in a P. Creating a list of the results to be used as possible bindings for the unbounded variable.

nof_instances	Number of instances in the task in the set of the P. After this function the number of instances that evaluated to invalid has been subtracted.
expr	The node for the start of the expression.
return	A list containing the result from each instance.

void evalUnbCmpExprP(tTree expr, symbol_t *instances, double_t **tmp_result_set_start, double_t **tmp_result_set_end)

Used by evalUnboundedCompareExprP to evaluate the expression an unbounded variable is compared to in a P.

tTree	Root node of the expression to evaluate.
instances	A list of all instance variables used in this expression.
tmp_result_set_start	A pointer to the start of the list were all values are to be put.
tmp_result_set_end	A pointer to the end of the list were all values are to be put.

double* evalUnbCmpExprPlt(tTree expr, symbol_t *probe_names, long *set_len)

Evaluates the expression an unbounded variable is compared to in a Plt (P with probes). Creating a list of the results to be used as possible bindings for the unbounded variable.

expr	The node for the start of the expression.
probe_names	List of all probes in this Plt.
set_len	Is increased once for every element in the returned list.
return	A list of all the results

evaluate_p.c -.h

Functions for evaluating P with task.

key functions

double loopFuncP(symbol_t *unbounded_instances, tTree node)

Creates nested loops for all unbounded instances in a P or a function subset on a task. Each loop sets a value for an unbounded instance. Only loops until values are found that make the expression true.	
unbounded_instances	A list of all unbounded instance variables.
node	The node for the function subset or P.
return	The result from the function if the subset/P. If a subset then the value from the function. If P then TRUE, FALSE or INVALID.

bool_t evalBoolP(tTree node)

Evaluates a boolean expression in a condition of a P with tasks.	
node	The root node of the expression to be evaluated
return	The result of the expression. TRUE, FALSE or INVALID

double evalNumP(tTree node)

Evaluates a numeric expression in the condition of a P with tasks.	
node	The root node of the expression to be evaluated.
return	The evaluated value

long evalInstanceP(tTree node)

Evaluates a instance expression.	
node	The root of the expression.
return	The evaluated value for the instance. This function could return negative values. Such invalid instances must be handled by the calling function.

evaluate_p_functions.c -.h

Contains functions to evaluate the statistical functions and the subset to file function for tasks.

evaluate_plt.c -.h

Functions for evaluating P on probes.

key functions

double loopFuncPlt(tTree node)

Evaluates a P with probes and calculates the probability of its condition being true. This is done by calling evalBoolPlt for all values the probes in the condition can have. The probability is calculated by dividing the time that the condition is fulfilled with the total time the probes involved are active.	
node	The node for a P with probes.
return	The probability of the condition in the P being true.

bool_t evalBoolPlt(tTree node)	
Evaluates a boolean probe expression.	
node	The root of the expression.
return	The result of the expression; TRUE or FALSE

double evalNumPlt(tTree node)	
Evaluates a numeric probe expression.	
node	The root of the expression.
return	The value of the expression.

evaluate_plt_functions.c -.h

Contains functions to evaluate the statistical functions and the subset to file function for probes.

boolean.h

Contains the enumeration bool_t with the values FALSE, TRUE and INVALID.

double.c -.h

A linked list of doubles. Contains structure and functions to add and remove elements as well as sorting etc.

error.c -.h

Contains error handling. Constants for various error messages as the enumeration error_t.

key global variables

bool_t has_errors	Set to TRUE when an error has been set.
error_t error_code	The error code of the last set error.

key functions

void setError(error_t code)	
Used throughout the application when an error has been found. Sets the error code and raises the error flag unless an error has already been set.	
code	The error code of the error we're setting.

log.c -.h

Contains structures for lists of tasks and probes. Functions for reading and compiling a trace and for accessing the lists.

key global variables

taskType task_list	The task list.
probelistType probe_list	The probe list.

key functions

int readLog(char* filename)

Reads all events from a log file.	
filename	Name of the log file.
return	1 if success, 0 otherwise (failed to open file)

bool_t compileEvents()

Compiles the event list into task and probe lists.	
return	FALSE if an error during compilation, TRUE otherwise.

double getTaskFieldValue(short task_id, long instance, unsigned char field_id, task_t** pTask, task_instance_t** pInstance, probe_t** pProbe)

Finds the value for a data member of an instance of a task. Updates the shortcut pointers to remember previous task, instance and probe to cut down on iterations as commonly this function is called for the same task repeatedly with consecutive instances.	
task_id	Id of the task.
instance	Index of the instance.
field_id	Id of the field.
pTask	A pointer to the task, possibly set by a previous run of this function. If not then it is set.
pInstance	A pointer to the instance, possibly set by a previous run of this function. If not then it is set.
pProbe	A pointer to the probe, possibly set by a previous run of this function. If not then it is set.
return	The value of the data member. If instance is not found then the invalid_value flag in the symbol table on top of the stack is set to TRUE.

long varNameToId(char* name)

Creates an id from a variable name using the ASCII values of the characters	
name	Name to turn into an id.
return	Id for the variable.

string2.c -.h

Contain a function for creating a string.

symbol.c -.h

Contain functions and structures for symbol tables.

key global variables

frst_symbol_table	The bottom of the symbol table stack.
last_symbol_table	The top of the symbol table stack.

key functions

void addSymbolTable(symbol_table_t *table)

Adds a symbol table to the symbol table stack.	
table	The symbol table to add.
first_symbol_table	If the stack was empty this is set to the table we add.
last_symbol_table	Is set to the table we add.

void removeSymbolTable()

Removes the symbol table on top of the table stack.	
first_symbol_table	Set to NULL if we removed the last table on the stack.
last_symbol_table	Moved to the previous table.

void addBoundedInstance(char *name)

Sets the bounded instance in the symbol table on top of the symbol table stack.	
name	Name to give the instance variable.
last_symbol_table	The bounded instance is reset and given the new name.

void addUnboundedInstance(char *name)

Adds an unbounded instance to the symbol table on top of the stack.	
name	Name of the instance variable to add.
last_symbol_table	The new unbounded instance is added if it did not already exist.

void addUnboundedVariable(char *name)

Sets the unbounded variable in the symbol table on top of the symbol table stack.	
name	Name to give the unbounded variable.
last_symbol_table	The unbounded variable is reset and given the new name.

void setSymbolInstanceValue(long variable_id, long value)

Sets a value for an instance variable (bounded or unbounded) in the symbol table on top of the symbol table stack.	
variable_id	Id of the variable to set.
value	The value to give the variable.

void setProbeValue(short probe_id, unsigned char value)

Sets the value of a probe in the symbol table on top of the symbol table stack.	
probe_id	Id of the probe to set a value for.
value	The value to set.

long getSymbolInstanceValue(long variable_id)	
Gets the current value of a symbol (a bounded or unbounded instance) in the symbol table on top of the stack.	
variable_id	Id of the instance variable.
return	The current value of the symbol.

unsigned char getProbeValue(short probe_id)	
Gets the current value of a probe in the symbol table on top of the stack.	
probe_id	Id of the probe.
return	The value of the probe.

tokentype.c -.h

Contain id constants for operators and functions used in tokens and then throughout the tool. Also some functions to check the type of an operator.

Appendix D. Test calculations

1. First two similar queries, in the P function we say that the response time of Task_FF should be between 25000000 and 75000000. In Test query D.1 we say that the probability of that should be greater than 0.75. In Test query D.2 we assign the true probability to the unbounded variable X. This is done following Semantic rule 3.31 and Semantic rule 3.14.

P(Task_FF(i), Task_FF(i).resp > 25000000 AND Task_FF(i).resp < 75000000) > 0.75

Test query D.1

P(Task_FF(i), Task_FF(i).resp > 25000000 AND Task_FF(i).resp < 75000000) = X

Test query D.2

Task_FF.resp	> 25 000 000	< 75 000 000	AND
912	FALSE	TRUE	FALSE
3 283 455	FALSE	TRUE	FALSE
72 298 761	TRUE	TRUE	TRUE
22 718 521	FALSE	TRUE	FALSE
41 080 759	TRUE	TRUE	TRUE
493 655 046	TRUE	FALSE	FALSE

Table D.1 truth table for the instances

In Table D.1 we can see that the condition is fulfilled for two out of the six instances giving us a probability of 2/6 or 0.333. As 0.333 is not greater than 0.75 the result from Test query D.1 should be FALSE. The result for Test query D.2 should be 0.333.

2. Here two queries where we in the P say that the combined response times of Task_FF and the following Task_TWO should be less than or equal to 75000000. This is done with combinations of Semantic rule 3.2, Semantic rule 3.5 and Semantic rule 3.14.

P(Task_FF(i), Task_FF(i).resp + Task_TWO(following(Task_FF(i))).resp <= 75000000) > 0.75

Test query D.3

P(Task_FF(i), Task_FF(i).resp + Task_TWO(following(Task_FF(i))).resp <= 75000000) = X

Test query D.4

First we find out what instance of Task_TWO follow each of the instances in Task_FF by comparing their end times as shown in Table D.2. Once that is done we simply add the two data members together and then make the comparison as in Table D.3.

Task_FF.end	Task_TWO.end	i	Task_TWO(following(Task_FF(i)))
4 596	5 486 546	0	0
31 800 000	86 315 613	1	1
204 754 884	555 550 564	2	2
644 218 521	1 000 564 654	3	3
901 235 413	1 068 452 486	4	3
1 562 107 532	2 225 541 232	5	5
	2 751 326 842		
	3 525 556 456		
	4 156 874 684		

Table D.2 calculating following instance

Task_FF(i).resp	Task_TWO(following(Task_FF(i))).resp	+	<= 75 000 000
912	5 486 423	5 487 335	TRUE
3 283 455	41 190 068	44 473 523	TRUE
72 298 761	460 445 019	532 743 780	FALSE
22 718 521	25 333 108	48 051 629	TRUE
41 080 759	25 333 108	66 413 867	TRUE
493 655 046	104 329 111	597 984 157	FALSE

Table D.3 calculating results for the instances

As we can see in Table D.3 this was true for four out of the six instances, i.e. a probability of 4/6 or 0.666. Thus the result for Test query D.3 should be FALSE and the result from Test query D.4 should be 0.666.

3. Here we have a test of unbounded instance variables using a pre-emption query. We say that no more than half of the instances of Task_FF may pre-empt Task_TWO. Here we use Semantic rule 3.8 and Semantic rule 3.32.

P(Task_FF(i), Task_FF(i).start > Task_TWO(j).start AND Task_FF(i).start < Task_TWO(j).end) <= 0.5

Test query D.5

First we list all the values, Table D.5 and Table D.4, and assign each of the instances in Task_TWO to a *j*. Then we make a truth table, Table D.6, for each of the expressions and for AND. The two expressions on their own are TRUE for all instances of Task_FF. It is however only for certain *j*. In order to get TRUE from in the AND column both must be TRUE for the same *j*.

Task_TWO.start	Task_TWO.end	j
123	5 486 546	0
45 125 545	86 315 613	1
95 105 545	555 550 564	2
975 231 546	1 000 564 654	3
1 000 580 000	1 068 452 486	4
2 121 212 121	2 225 541 232	5
2 525 352 525	2 751 326 842	6
2 845 699 994	3 525 556 456	7
4 026 430 015	4 156 874 684	8

Table D.4 list values for the unbounded instance variable

Task_FF.start	i
3 684	0
28 516 545	1
132 456 123	2
621 500 000	3
860 154 654	4
1 068 452 486	5

Table D.5 list values for the bounded instance variable

Task_FF(i).start > Task_TWO(j).start	Task_FF(i).start < Task_TWO(j).end	AND
TRUE (j < 1)	TRUE (j >= 0)	TRUE (j = 0)
TRUE (j < 1)	TRUE (j > 0)	FALSE
TRUE (j < 3)	TRUE (j > 1)	TRUE (j = 2)
TRUE (j < 3)	TRUE (j > 2)	FALSE
TRUE (j < 3)	TRUE (j > 2)	FALSE
TRUE (j < 5)	TRUE (j > 4)	FALSE

Table D.6 result for each instance

This was true for two out of the six instances. Thus the result for the query should be TRUE as 0.333 is less than or equal to 0.5.

4. Here we calculate an average function on probe30.

avg(*.probe30)

Test query D.6

First we list all events for this probe, the time and data of the event. By subtracting the start time from the start time of the following event we get the time the probe had each value. The last value is held until the end of the trace which in this case is the time 4294967295. We multiply each value with the time it was held and then sum up these values. This gives us the sum of the value for each time unit. We calculate the total time this probe has values as the end time – the start time of the first event. The average value for this probe is found by dividing the sum of data with the total time.

data	start time	duration time	data * duration time
1	302	50 486 241	50 486 241
6	50 486 543	49 519 121	297 114 726
156	100 005 664	879 306 990	137 171 890 440
255	979 312 654	21 487 345	5 479 272 975
139	1 000 799 999	3 294 167 296	457 889 254 144
	4 294 967 295		
		sum:	600 888 018 526
		total time:	4 294 966 993
		avg:	139,905154

Table D.7 calculating a probe

5. Here we have a P where we say that probe30 should have a value greater than 100 and at the same time probe255 should have a value less than 60. The probability of this we will bind to the unbounded variable X. This is done with a combination of Semantic rule 3.12, Semantic rule 3.14 and Semantic rule 3.31.

P(*, *.probe30 > 100 AND *.probe255 < 60) = X

Test query D.7

We begin with listing all events, data and timestamps, for both probes in Table D.8. In Table D.9 we then merge the two event list so that we can see what value each probe had at all the times. The first event for probe255 occur at time 4 308. Thus it does not have any value when probe30 gets its first value at time 302. The time before 4 308 must be excluded. We make the comparisons to find out during what times values was set that makes the expression true.

probe30 data	probe30 start time	probe255 data	probe255 start time
1	302	40	4 308
6	50 486 543	60	70 567 456
156	100 005 664	10	625 048 658
255	979 312 654	20	1 070 211 330
139	1 000 799 999	60	2 876 700 005
		60	3 964 446 434

Table D.8 list all values

time	probe30	probe255	probe30 > 100	probe255 < 60	AND
302	1	-	FALSE	-	-
4 308	1	40	FALSE	TRUE	FALSE
50 486 543	6	40	FALSE	TRUE	FALSE
70 567 456	6	60	FALSE	FALSE	FALSE
100 005 664	156	60	TRUE	FALSE	FALSE
625 048 658	156	10	TRUE	TRUE	TRUE
979 312 654	255	10	TRUE	TRUE	TRUE
1 000 799 999	139	10	TRUE	TRUE	TRUE
1 070 211 330	139	20	TRUE	TRUE	TRUE
2 876 700 005	139	60	TRUE	FALSE	FALSE
3 964 446 434	139	60	TRUE	FALSE	FALSE
4 294 967 295					

Table D.9 truth table

We calculate the number of total time units it was true: (979312654 - 625048658) + (1000799999 - 979312654) + (1070211330 - 1000799999) + (2876700005 - 1070211330) = 2251651347. We calculate the total time that both probes had values as the last timestamp of the log minus the first timestamp when both have value: 4294967295 – 4308 = 4294962987. Finally we calculate the probability as 2251651347 / 4294962987 = 0.524. So the result for this query should be to bind X to 0.524.

6. A calculation for an inner unbounded variable. In the query we ask what deadlines will Task_FF miss with a probability greater than 0.75.

$$P(\text{Task\_FF}(i), \text{Task\_FF}(i).\text{resp} > X) > 0.75$$

Test query D.8

We begin by listing all the values we are to try as values for the unbounded variable. In this query those values are simply all the response times of Task_FF. We sort those values and then compare each of them with each instance. Then counting those who are TRUE we find the probability for each binding. Comparing these probabilities we find out what values makes the query true. In this case only the first value did that. Of the probabilities only 0.833 is greater than 0.75. According to the semantic rule for this query, Semantic rule 3.20, the interval should be from -8 up to the smallest of the values that did not make the query true, i.e. (-8..3283455).

X	Task_FF.resp						Probability
	912	3 283 455	72 298 761	22 718 521	41 080 759	493 655 046	
912	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	5/6 = 0.833
3 283 455	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE	4/6 = 0.666
22 718 521	FALSE	FALSE	TRUE	FALSE	TRUE	TRUE	3/6 = 0.500
41 080 759	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	2/6 = 0.333
72 298 761	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	1/6 = 0.166
493 655 046	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	0/6 = 0.000

Table D.10 truth table for all tested values

7. A calculation for another inner unbounded variable. Here we have an outer strict equal. The query asks for the deadlines that all instances of Task_FF will meet with a margin of 5000.

P(Task_FF(i), Task_FF(i).resp + 5000 <= X) = 1

Test query D.9

We begin, in Table D.11, by listing all the values we are to try as values for the unbounded variable. In this case those are the response times + 5000. We sort those values and then compare each of them with each instance. Then counting those who are TRUE we find the probability for each binding. Here we find that we have one value that makes the query true, 493660046. As this value is the last in the set we have encounter one of the special cases in assigning. We must now find out if this value is the only valid binding or if greater values also work. To do so we try for 493660046 + 1 = 493660047. As shown in Table D.12 that value we also get the probability 1. Hence the interval of deadlines that will always be met with a margin of 5000 is [493660046..8). Here we used Semantic rule 3.3 and Semantic rule 3.27

X	Task_FF.resp + 5 000						Probability
	5 912	3 288 455	72 303 761	22 723 521	41 085 759	493 660 046	
5 912	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	1/6 = 0.166
3 288 455	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	2/6 = 0.333
22 723 521	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE	3/6 = 0.500
41 085 759	TRUE	TRUE	FALSE	TRUE	TRUE	FALSE	4/6 = 0.666
72 303 761	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	5/6 = 0.833
493 660 046	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	6/6 = 1.000

Table D.11 truth table for the tested values

493 660 047	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	6/6 = 1.000

Table D.12 truth table for the extra tested value

8. One more test with an inner unbounded variable. Here we want the deadlines that half of the instances will meet.

P(Task_FF(i), Task_FF(i).resp <= X) = 0.5

Test query D.10

Like before we list the bindings and make the comparisons. Here we have one value that makes the query true, 22718521. Looking at the semantic rule for a query like this, Semantic rule 3.27, we see that we should create an interval from the least of the values that makes it true up to the least of the greater values that make it false. In this case the interval would be [22718521..41080759).

X	Task_FF.resp						Probability
	912	**3 283 455**	**72 298 761**	**22 718 521**	**41 080 759**	**493 655 046**	
912	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	1/6 = 0.166
3 283 455	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	2/6 = 0.333
22 718 521	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE	3/6 = 0.500
41 080 759	TRUE	TRUE	FALSE	TRUE	TRUE	FALSE	4/6 = 0.666
72 298 761	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	5/6 = 0.833
493 655 046	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	6/6 = 1.000

Table D.13 truth table for the tested values

9. In Test query D.11 we calculate an inner unbounded variable with an inner strict equal. We ask for the exact deadlines that are met with a probability of 0.1.

P(Task_FF(i), Task_FF(i).resp = X) > 0.1

Test query D.11

Like always when calculating inner unbounded variables we list the bindings and make the comparisons. As no instances have the same response times in this example the P will be true for one instance for each value, i.e. when we compare a response time with it self. Thus all values will give us a probability of 0.166. 0.166 is greater than 0.1 so all values are true. According to Semantic rule 3.30 we should not create an interval but a set containing all values that made the query true. Hence we bind X to the set [912, 3283455, 72298761, 22718521, 41080759, 493655046].

X	Task_FF.resp						Probability
	912	3 283 455	72 298 761	22 718 521	41 080 759	493 655 046	
912	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	1/6 = 0.166
3 283 455	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	1/6 = 0.166
22 718 521	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	1/6 = 0.166
41 080 759	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	1/6 = 0.166
72 298 761	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	1/6 = 0.166
493 655 046	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	1/6 = 0.166

Table D.14 truth table for the tested values

Appendix E. User Guide

The program takes three parameters. First the name of the trace to analyse, secondly the name of the query file and finally the name of the result file. The query and result file are optional. If not given the standard files query.ppl and result.ppl will be used. The query file contains the PPL queries to evaluate. If it contain several queries they should be separated with ';'. The result file is where the results from the queries will be written.

```
PPL trace.log query_file.txt result_file.txt
```

Example E.1 starting the program

There are two kinds of queries: a property or a single function. There are two kinds of functions that can be used as queries. The statistical functions and the subset function.

A property is a comparison between two probabilities. A probability is a P function, a constant or an unbounded variable.

A P function calculates the probability of an instance of a set, a task, fulfilling a condition. It takes two arguments. The first is the working set and the second is the condition. The working set argument is the name of the task and an *instance operator*. The variable in this instance operator is the one and only bounded instance variable of the query. The condition is a Boolean expression.

```
P(T1(i), <condition>)
```

Example E.2 P function

A task instance has five data members. The start time, *start*, the end time, *end*, the response time, *resp*, and the execution time, *exec*. Finally the name of a probe can be used as a data member to get the value that probe had when the instance begun its execution.

PPL have five relational operators. Greater than '>', greater than or equal '>=', strict equal '=', less than or equal '<=' and less than '<'. All four basic arithmetic operators, '+', '-', '*' and '/' can be used. The absolute value function *abs* and parenthesises are also allowed in arithmetic expressions. There are three logical operators. The binary AND and OR and the unary NOT.

The instance operator binds instances of the task it is part of. It consists of a variable to represent the instance it is currently bounded to. To compare relative instances a numeric value can be added or subtracted to this variable. Here for example we calculate the probability of two consecutive instances of the task T1 having response times greater than 50.

```
P(T1(i), T1(i).resp > 50 AND T1(i+1).resp > 50)
```

Example E.3 relative instances

A P like that on several consecutive instances can be simplified by adding or subtracting a sequence instead of a single numeric value. Here we use a sequence to calculate the probability of three consecutive instances having a response time greater than 50.

```
P(T1(i), T1(i+[0..2]).resp > 50)
```

Example E.4 sequencing

When comparing different tasks we can use the function following to find relative instances. Following finds the instance of a task that execute closest after some other instance. For example we can say that T1 and the closest following T2 should have a combined response time of less than 500.

```
P(T1(i), T1(i).resp + T2(following(T1(i))).resp < 500)
```

Example E.5 following

If the variable of an instance operator is not the bounded instance variable then it is an *unbounded instance variable*. While the bounded instance variable will be bounded to all instances of the task in the set of the P, an unbounded instance is only bounded once for each instance in that set. That one binding should be such that the condition is fulfilled. For example we can calculate the probability of the execution times of T1 being greater than some execution time for T2.

```
P(T1(i), T1(i).exec > T2(j).exec)
```

Example E.6 unbounded instance variable

The unbounded instance variables are useful for example when calculating probabilities of pre-emption. The probability of T1 being pre-empted by T2 could be formulated as the probability of some instance of T2 starting after the start and before the end of a T1.

```
P(T1(i), T2(j).start > T1(i).start AND T2(j).start < T1(i).end)
```

Example E.7 pre-emption

The working set of a P could also be the union of all tasks, '*'. With this set no single task can be used in the condition and only probe data members can be accessed. Furthermore the instance

operator can not be applied to this set. That is because with this set the probability is not calculated on instances but over time. Normally the probability is calculated as the number of instances that fulfil the condition divided by the total number of instances. Here the instances are replaced by time units. For example we could calculate the probability of probe20 always being greater than 0 as the number of time units it is greater than 0 divided by the total number of time units it has a value.

```
P(*, *.probe20 > 0)
```

Example E.8 P on probes

Unbounded variables can be used as either a probability or as part of a condition in a P. Normally a property is evaluated as true or false. If the property contains an unbounded variable the result will instead be the interval of values that, in the place of the variable, make the property true. For example we can use an unbounded variable X as the probability of T1 having a response time greater than 50.

```
P(T1(i), T1(i).resp > 50) = X
```

Example E.9 unbounded probability

Placed in the condition the unbounded variable could for example be used to find what deadline T1 would meet with a probability of at least 0.8.

```
P(T1(i), T1(i).resp < X) >= 0.8
```

Example E.10 inner unbounded variable

There are four statistical functions in PPL: *min*, *max*, *avg* and *median*. They take as argument the set of data it is to be applied to. The set is written as the name of the task and what data member of that task. Like with the P function the set '*' can be used to calculate on a probe over time. For example we can calculate the average response time of T1.

```
avg(T1.resp)
```

Example E.11 statistical function

Like the P function the statistical functions can take a condition as an argument. In that case the function will be applied to the subset of the set that fulfil the condition. For example the minimum of the response times who are greater than 50.

```
min(T1(i).resp, T1(i).resp > 50)
```

Example E.12 statistical function with a condition

Those four statistical functions can be used alone as queries or they can be part of a condition in a P or another statistical function. For example we could calculate the probability of the response times of T1 being greater than the average response time of T2.

```
P(T1(i), T1(i).resp > avg(T2.resp))
```

Example E.13 statistical function in the condition of a P

Like the statistical functions the subset function takes a set and a condition as arguments. But instead of performing any calculation on the subset that fulfils the condition it writes all the values of that subset to a file. It is possible to leave the condition argument out to write all values from a set to a file. We can for example write all response times of T1 that are greater than 50 to the file "T1_resp.txt". The greater than operator is in this context used as a pipe between the function and the filename. If the set is a probe, *.probeX, then the values of that probe and the number of time units the probe had that value is written.

```
subset(T1(i).resp, T1(i).resp > 50) > "T1_resp.txt"
```

Example E.14 the subset function

Finally we list the different error messages. The error code is written in the result file. For some errors it is impossible to continue when they occur. In those cases only one error code will be written to the result file. Other errors only affect the query that causes them. In those cases the result from the rest of the queries will be written as usual while the result from the faulty query will be the error code. For name errors the faulty name will be written along with the error code. Furthermore there are several other errors that can only occur as a result of bugs and thus should never happen.

Error code	ERROR 2
Error name	ERR_DIVISION_BY_ZERO
Description	An arithmetic expression resulted in a division by zero.

Error code	ERROR 10
Error name	ERR_EMPTY_SET
Description	A set of instances is empty. Typically because of a statistical function with a condition where no instance fulfilled the condition.

Error code	ERROR 11
Error name	ERR_NO_PROBES
Description	The condition of a function with the working set '*' does not contain a probe.

Error code	ERROR 16
Error name	ERR_NO_VALID_BINDINGS
Description	There where no values to try as bindings for an unbounded variable.

Error code	ERROR 18
Error name	ERR_NO_QUERY
Description	The query file was not found.

Error code	ERROR 19
Error name	ERR_ILLEGAL_SEQUENCE
Description	Illegal values in a sequence. (Must be ascending.)

Error code	ERROR 20
Error name	ERR_TASK_IN_PROBE_QUERY
Description	The condition of a P with the working set * contains a task.

Error code	ERROR 21
Error name	ERR_PROBE_IN_TASK_QUERY
Description	The condition of a P with a single task as working set contains the construct *.probeX.

Error code	ERROR 22
Error name	ERR_PARSE_ERROR
Description	A syntactical error in a query.

Error code	ERROR 23
Error name	ERR_UNBOUNDED_IN_SUBSET
Description	An unbounded variable in the condition of some function other than P.

Error code	ERROR 24
Error name	ERR_ILLEGAL_PROBE
Description	An illegal probe id. Must be between 16 and 255.

Error code	ERROR 25
Error name	ERR_TOO_MANY_UNBOUNDED
Description	More than one unbounded variables in a query.

Error code	ERROR 26
Error name	ERR_NAME_ERROR
Description	A task or probe name does not exist in the trace or a illegal variable name. Variables must be alphanumeric (plus '_') and no more than four characters long. Note that the name check is case sensitive.

Error code	ERROR 27
Error name	ERR_COMPILE_ERROR
Description	An error occurred while compiling the events from the trace.

Error code	ERROR 28
Error name	ERR_READ_LOG_ERROR
Description	Failed to open the trace. Because the file does not exist or is of an unknown version.

Error code	ERROR 29
Error name	ERR_INVALID_PROBABILITY
Description	A probability constant is greater than 1 or less than 0 or such a comparison.

Error code	ERROR 30
Error name	ERR_TYPE_ERROR_P
Description	The condition in a P is of wrong type. (Should be boolean.)

Error code	ERROR 31
Error name	ERR_TYPE_ERROR_NOT
Description	The operand of a NOT operator is of wrong type. (Should be boolean.)

Error code	ERROR 32
Error name	ERR_TYPE_ERROR_LOGOP
Description	The operand of a binary logical operator is of wrong type. (Should be boolean.)

Error code	ERROR 33
Error name	ERR_TYPE_ERROR_ARITOP
Description	The operand of an arithmetic operator is of wrong type. (Should be numeric.)

Error code	ERROR 35
Error name	ERR_TYPE_ERROR_FUNCTION_SUBSET
Description	The condition of a statistical function is of wrong type. (Should be boolean.)

Error code	ERROR 36
Error name	ERR_TYPE_ERROR_UMINUS
Description	The operand of a unary minus operator is of wrong type. (Should be numeric.)

Error code	ERROR 37
Error name	ERR_TYPE_ERROR_RELOP
Description	The operand of a relational operator is of wrong type. (Should be numeric.)

Error code	ERROR 39
Error name	ERR_NO_PROBE_TIME
Description	One of the used probes has a value for 0 time units.