Natural Language Specification of Performance Trees

Lei Wang, Nicholas J. Dingle, and William J. Knottenbelt

Department of Computing, Imperial College London,
180 Queen's Gate, London SW7 2BZ, United Kingdom.
Email: {lw205,njd200,wjk}@doc.ic.ac.uk

Abstract. The accessible specification of performance queries is a key challenge in performance analysis. To this end, we seek to combine the intuitive aspects of natural language query specification with the expressive power and flexibility of the Performance Tree formalism. Specifically, we present a structured English grammar for Performance Trees, and use it to implement a Natural Language Query Builder (NLQB) for the Platform Independent Petri net Editor (PIPE). The NLQB guides users in the construction of performance queries in an iterative fashion, presenting at each step a range of natural language alternatives that are appropriate in the query context. We demonstrate our technique in the specification of performance queries on a model of a hospital's Accident and Emergency department.

Key words: Performance requirements specification; Natural language; Performance Trees; Performance analysis

1 Introduction

Performance is a vital consideration for system designers and engineers. Indeed, a system which fails to meet its performance requirements can be as ineffectual as one which fails to meet its correctness requirements. Ideally, it should be possible to determine whether or not this will be the case at design time. This can be achieved through the construction and analysis of a performance model of the system in question, using formalisms such as queueing networks, stochastic Petri nets and stochastic process algebras.

One of the key challenges in performance analysis is to provide system designers with an accessible yet expressive way to specify a range of performance-related queries. These include *performance measures*, which are directed at numerical performance metrics (e.g. *"In a hospital, what is the utilisation of the operating theatre?"*), and *performance requirements*, which indicate conformity to a QoS constraint (e.g. *"In a mobile communications network, is the time taken to send an SMS message between two handsets less than 5 seconds with more than 95% probability?"*).

Formalisms such as Continuous Stochastic Logic (CSL) [3, 4] provide a concise and rigorous way to pose performance questions and allow for the composition of simple queries into more complex ones. Such logics can be somewhat daunting for non-expert users; indeed, a study by Grunkse [9] found that industrial users attempting to specify requirements sometimes put forward formulae that were syntactically incorrect. Even for those comfortable with their use, there still remains the problem of correctly converting informally-specified requirements into logical formulae. Further, CSL is limited in its expressiveness, since it is unable to reason about certain concepts such as higher moments of response time.

Performance Trees [14, 15] were recently proposed as a means to overcome these problems. These are an intuitive graphical formalism for expressing performance properties. The concepts expressible in Performance Tree queries are intended to be familiar to engineers and include steady-state measures, passage time distributions and densities, their moments, action frequencies, convolutions and arithmetic operations. An important concern during the development of Performance Trees was ease of use, resulting in a formalism that can be straightforwardly visualised and manipulated as hierarchical tree structures.

Another approach for the accessible specification of performance queries is the use of natural language, whereby users specify their queries textually before they are automatically translated into logical formulae. This allows users to exploit the power of logical formalisms without requiring in-depth familiarity and also minimises the chances of misspecification. Prior work has focused on both unstructured [10] and structured [8, 9, 12, 13] natural language query specification, albeit mostly in the context of correctness – rather than performance – analysis.

Unstructured natural language specification allows a user to freely enter sentences which must then be parsed and checked before being converted into a corresponding performance property. Although this is perhaps the most intuitive query specification mechanism, it must incorporate strategies for resolving ambiguities and context-specific expressions. The conversion process is therefore often iterative, with the user refining their natural language expression in response to the checking until it can be successfully converted into a property.

By contrast, structured natural language specification presents users with a set of expressions which can be composed together in accordance with a predefined structured grammar. If the same grammar is also defined for the logic into which the query will be converted (e.g. as in [9]), the conversion process is relatively straightforward. The main advantage of such structured specification is therefore that there is less "trial and error" involved in forming a query: the user's choices are limited to those provided by the grammar and so they can only construct a natural language query which will always convert directly into a logical formula.

In this paper, we present a structured natural language query specification mechanism for Performance Trees to further improve their accessibility. The grammar of this structured mechanism is provided by the syntax of Performance

Trees, which enables a structured natural language query to be converted into a Performance Tree and then evaluated using the existing Performance Tree evaluation architecture [6]. Furthermore, taken together, the natural language and Performance Tree representations provide mutual validation, allowing the user to ensure that their queries capture exactly the performance properties of interest.

The rest of this paper is organised as follows. Section 2 provides a brief overview of Performance Trees and the tool support for their evaluation. Section 3 then presents our structured grammar for the natural language representation of Performance Trees and describes its implementation within the Natural Language Query Builder (NLQB), a module for the Platform Independent Petri net Editor (PIPE) [1, 5]. Section 4 demonstrates the use of the NLQB in a case study of a hospital's Accident and Emergency unit. Section 5 concludes and discusses future work.

2 Performance Trees

Performance Trees [14, 15] are a formalism for the representation of performance-related queries. They combine the ability to specify performance requirements – i.e. queries aiming to determine whether particular properties hold on system models – with the ability to extract performance measures – i.e. quantifiable performance metrics of interest.

A Performance Tree query is represented as a tree structure consisting of nodes and interconnecting arcs. Nodes can have two kinds of roles: *operation* nodes represent performance-related functions, such as the calculation of a passage time density, while *value* nodes represent basic concepts such as a set of states, an action, or simply numerical or Boolean constants.

Complex queries can be easily constructed by connecting operation and value nodes together. The formalism also supports macros, which allow new concepts to be created with the use of existing operators, and an abstract state-set specification mechanism to enable the user to specify groups of states relevant to a performance measure in terms of the corresponding high-level model (whether this be a stochastic Petri net, queueing network, stochastic process algebra etc.)

Performance Trees have been fully integrated into the Platform Independent Petri net Editor (PIPE), thus allowing users to design Generalised Stochastic Petri Net (GSPN) [2] models and to specify relevant performance queries within a unified environment. PIPE communicates with an Analysis Server which employs a number of (potentially parallel and distributed) analysis tools [7, 11] to calculate performance measures. These include steady-state measures, passage time densities and quantiles, and transient state distributions.

Performance Tree Node	Natural Language Representation	Arguments	Output
RESULT	"is it true that"	*InInterval* | *Subset* | ¬ | ∧ / ∨ | ≥, >, ==, <, ≤	N/A
	"what is the"	*PTD* | *Dist* | *Conv* | *Moment* | *SS:P* | *SS:S* | *FR* | *ProbInInterval* | *ProbInStates* | *StatesAtTime* | ⊕	N/A
PTD	"the passage time density defined by start states" **states** "and target states" **states**	**states**, **states**	*PTD*
Dist	"the cumulative distribution function calculated from" *PTD*	*PTD*	*Dist*
Conv	"the convolution of" *PTD* "and" *PTD*	*PTD*, *PTD*	*PTD*
SS:P	"the steady-state probability distribution of" **statefunc** "applied over" **states**	**statefunc**, **states**	**num**
Perctl	"the" **num** "percentile of" *PTD* | *Dist*	**num**, *PTD* | **num**, *Dist*	**num**
StatesAtTime	"the set of states that the system can be in at the time instant" **num** "within probability bound" *Range*	**num**, *Range*	**states**
ProbInStates	"the transient probability of the system having started in" **states** "and being in" **states** "at the time instant given by" **num**	**states**, **states**, **num**	**num**
Moment	"the" **num** " raw moment of" *PTD* | *Dist*	**num**, *PTD* | **num**, *Dist*	**num**
FR	"the frequency of" **action**	**action**	**num**
ProbInInterval	"the probability with which a value sampled from" *PTD* "lies within" *Range*	*PTD*, *Range*	**num**
InInterval	**num** "lies within" *Range*	**num**, *Range*	**bool**
Subset	**states** "is a subset of" **states**	**states**, **states**	**bool**
∧ / ∨	**bool** "and/or" **bool** "holds"	**bool**, **bool**	**bool**
¬	"the negation of" **bool** "holds"	**bool**	**bool**
≥, >, ==, <, ≤	**num** "greater than or equal to/greater than/equal to/less than/less than or equal to" **num**	**num**, **num**	**bool**
⊕	**num** "plus/minus/raised to the power of/ multiplied by/divided by" **num**	**num**, **num**	**num**
Range	"the range" **num** "to" **num**	**num**, **num**	**num**

Table 1. Structured grammar for Performance Trees.

Node	Description
action	The name of an action (transition in GSPN context)
bool	True or False
num	A real number
states	A specification of a subset of reachable states
statefunc	A function applied to a state that returns a real number

Table 2. Description of user-specified value nodes

3 Structured Grammar for Performance Tree Specification

The current Performance Query Editor incorporated into PIPE requires users to be familiar with Performance Tree nodes (including their graphical representations and semantics). Because of this, a "drag and drop" graphical approach to building a Performance Tree query can be quite time-consuming. We have therefore developed an alternative approach based on structured natural language and implemented this in the Natural Language Query Builder (NLQB). The NLQB enables users to build performance queries in an iterative manner by selecting natural language fragments from a constantly-updated pull-down menu.

As shown in Table 1, the foundation of the NLQB is a structured natural language grammar derived from the syntax of Performance Trees. Following the convention introduced in [9], non-terminals (operation nodes) are shown in *italics*, literal terminals (the natural language representation) are given in quotation marks ("") and non-literal terminals are given in **bold**. These non-literal terminals are user-supplied value nodes and can only be of type **num**, **bool**, **states**, **statefunc** and **action**. A description of the permitted values for these nodes is given in Table 2.

3.1 Using the Natural Language Query Builder

Fig. 1 shows the NLQB in use. The user selects the appropriate phrases from the drop-down menu underneath the main graphical display and at the same time the corresponding Performance Tree is automatically constructed. When a selection has been made, the selected phrase is inserted in the natural language query in the text area and, at the same time, a corresponding Performance Tree node is plotted in an appropriate position. An automatic positioning mechanism calculates the coordinates of the recently created node and its outgoing arcs according to the position of its parent node and its level in the tree. The position of nodes and arcs can be adjusted manually if the user is not satisfied with the automatic positioning.

Each option in the drop-down menu consist of two elements – the natural language representation and the expected arguments. The natural language representation explains the operation that the node carries out, and the expected arguments (displayed in square brackets) indicate the type of its child nodes. The first expected argument is coloured in red, and all other expected arguments are coloured in blue. The user then specifies arguments in turn via the drop-down menu. As an argument is specified, its natural language representation is added to the query. When a value node is required, a dialog is presented to allow the user to make the required assignment.

For example, the *InInterval* node is expressed as "*num* lies within *Range*". When it is selected by the user, the first expected argument, *num*, indicates that a numerical value is required as input, so the NLQB uses the structured natural

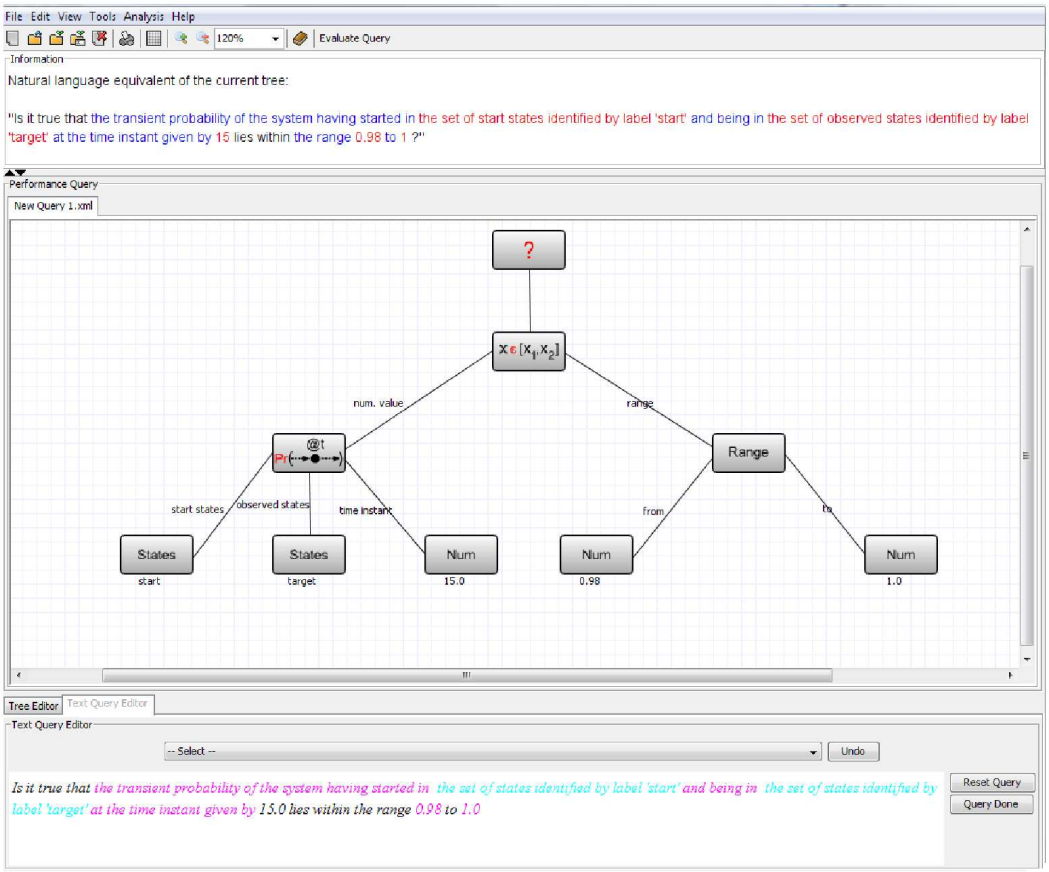

Fig. 1. Screenshot of the Natural Language Query Builder, showing a natural language query specification and the corresponding Performance Tree.

language grammar (given in Table 1) to find all nodes that produce numerical output and inserts their natural language representation into the drop-down menu. The other expected argument is a *Range* node; the NLQB only displays the corresponding phrase "the range *num* to *num*" in the menu after the first argument to the *InInterval* node has been supplied.

Each natural language phrase is presented in a different colour according to the output type of the node it represents. For example, phrases representing nodes with Boolean output are coloured black but phrases representing a set of states are in cyan. This aids readability by helping users to easily categorise each part of the natural language representation of their query. The NLQB also provides an undo mechanism to allow users to correct their query.

4 Case Study

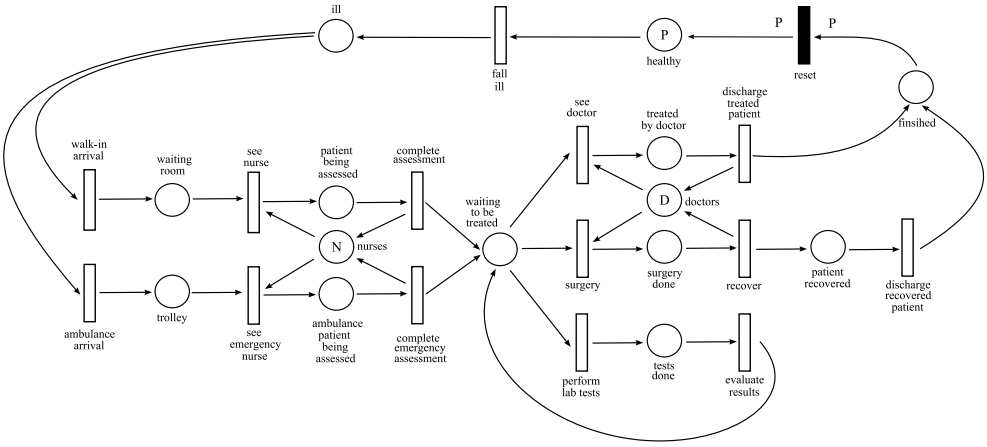

Fig. 2. GSPN model of a Hospital A&E Department [6]

We demonstrate how to design queries and calculate the relevant results using the NLQB for two examples based on the Accident and Emergency (A&E) department GSPN model of [6] shown in Fig. 2. There is an initial group of healthy people who fall ill and go to a hospital – arriving either by walking in or by ambulance. Walk-in patients wait in the waiting room for assessment until a nurse becomes available, while ambulance patients wait on a trolley to be assessed by a nurse. Patients are subsequently either seen by a doctor for treatment, sent for lab tests or sent for surgery. The model is parameterised by P, N and D, which denote the number of tokens on the places *healthy* (people), *nurses* and *doctors*, respectively. In the following examples, we set $P = 10$, $N = 4$ and $D = 4$, yielding an underlying Markov chain with 313 986 states.

Example 1 We wish to answer the performance query:

> What is the cumulative distribution function of the time taken for all patients in the system to fall ill, complete treatment and be discharged from the hospital?

The first thing the NLQB needs to know is whether this query expects a truth value or a quantitative measure as its result. Therefore, the only two available options in the drop-down menu are "Is it true that [bool]?" and "What is the [quantitative measure]?" In this case, we select the second option.

As we have selected the quantitative measure option, the NLQB interrogates the structured grammar table, extracts all operations that produce quantitative values and places their natural language representations into the drop-down

menu. We are interested in computing a passage time distribution and so we choose the "cumulative distribution function calculated from [PTD]" as our next input. This is incorporated into the natural language representation of the query and at the same time a *Dist* node is created in the Performance Tree and connected to the *RESULT* node.

The next argument to be specified is "[PTD]", which is displayed in red. This requires two sets of states as arguments which are assigned manually when "Assign States" is selected from the menu (using PIPE's state assignment tool). The specification of the start states in this query (in this case a single start state) is given as:

$$\text{all patients healthy} := (\#(\text{healthy}) = 10) \wedge (\#(\text{nurses}) = 4) \wedge (\#(\text{doctors}) = 4)$$

Similarly, the specification of the target states is:

$$\text{all patients treated} := (\#(\text{finished}) = 10)$$

where $\#(p)$ returns the number of tokens on place p in the model.

The completed query is shown in Fig. 3. The resulting natural language specification is:

> What is the cumulative distribution function calculated from the passage time density defined by the set of start states 'all patients healthy' and the set of target states 'all patients treated'?

Fig. 4 shows the result of evaluating the PTD (passage time density) node subquery, while Fig. 5 shows the cumulative distribution function resulting from the evaluation of the overall query.

Example 2 We wish to answer the performance query:

> What is the probability that all patients complete treatment and are discharged from the hospital within 4 time units?

This is constructed in a similar way as Example 1 and, as shown in Fig. 6, the NLQB produces the following natural language specification:

> What is the probability with which a value sampled from the passage time density defined by the set of start states 'all patients healthy' and the set of target states 'all patients treated' lies within the range 0 to 4?

From the cumulative distribution function in Fig. 5, we can see that the probability that all patients complete their treatment within 4 time units is 0.933 (rounded to 3 decimal places).

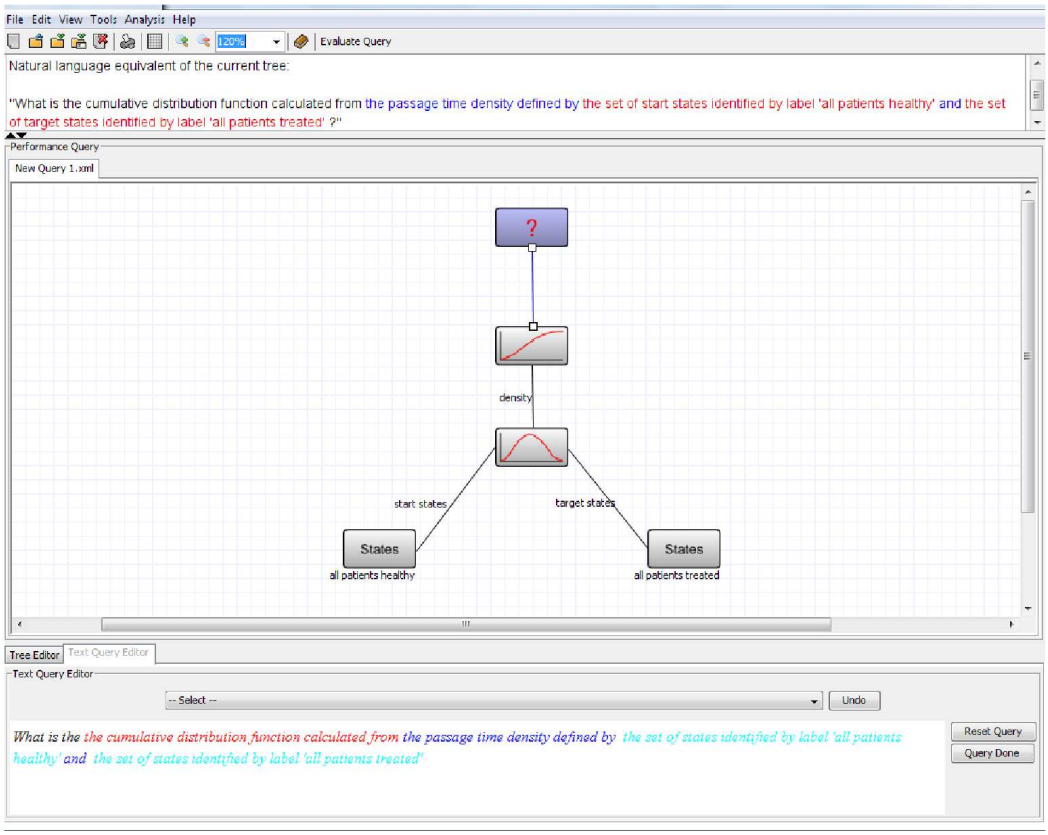

Fig. 3. The expression of Example 1 in the NLQB.

5 Conclusion

In this paper, we have presented a structured natural language query specification mechanism for Performance Trees. We have implemented this in PIPE as the Natural Language Query Builder which can be used with existing analysis tools to specify and calculate performance measures of interest.

There are a number of avenues for future work. Firstly, we are working to provide support for queries tailored to specific user models, i.e. support for model-specific terminology that takes into account the semantic meaning of model components. For example, in the context of the A&E model, we would like to be able to input queries such as "Is the time from the first patient to fall ill to the time of discharge from the hospital less than 4 hours at least 98% of the time?" We intend to accomplish this by requiring the user to augment the system model with information relating abstract model components to real world entities (e.g. in the context of a Petri model, what do the tokens on particular places repre-

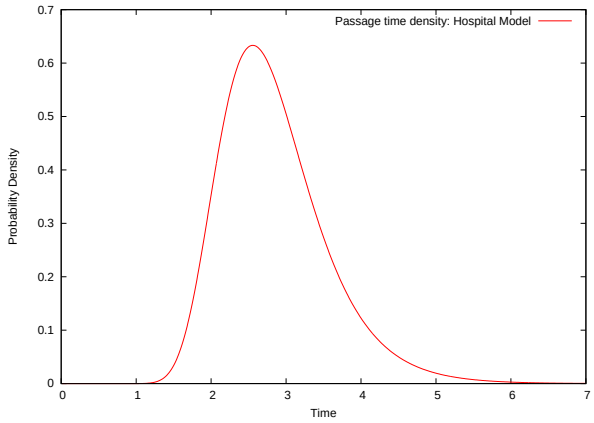

Fig. 4. Probability density function of the time taken to process all patients in the hospital model.

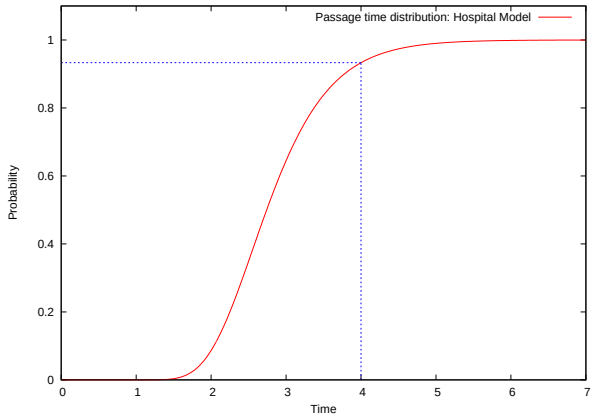

Fig. 5. Cumulative distribution function of the time taken to process all patients in the hospital model. The probability at time $t = 4$ is also marked.

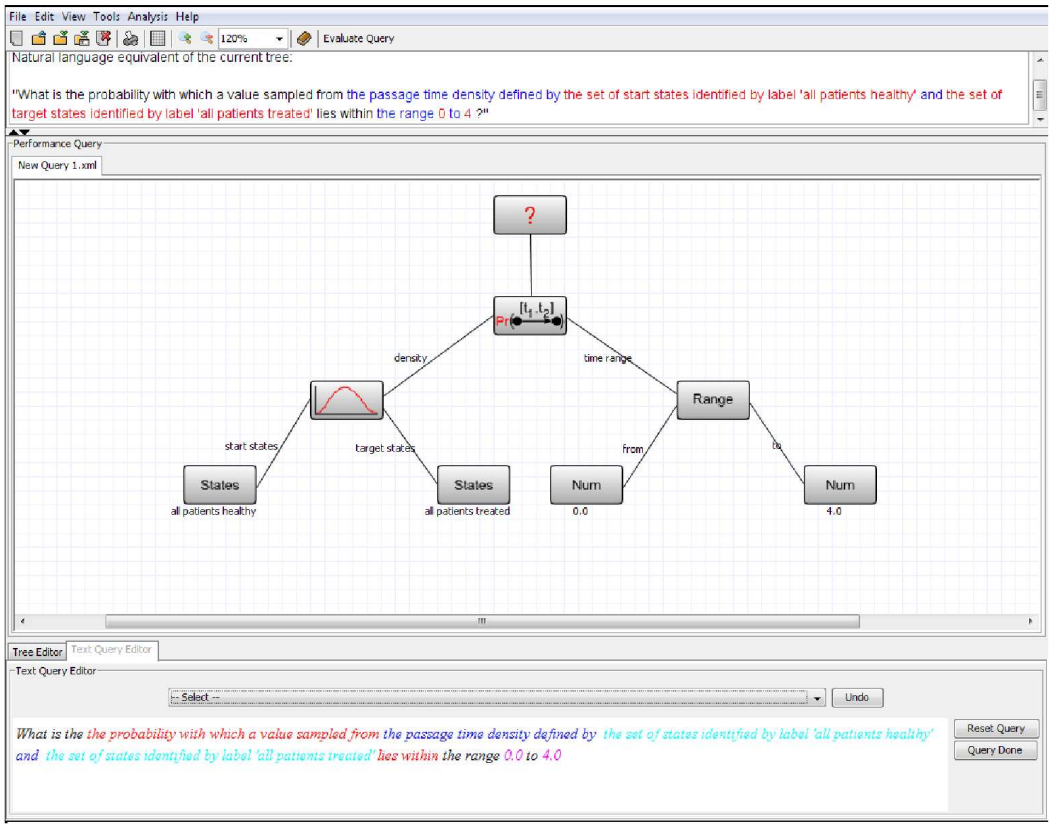

Fig. 6. The expression of Example 2 in the NLQB.

sent?) Secondly, we would like to augment the Performance Tree formalism with an experimental framework so that we can pose questions such as "How many doctors should be employed to ensure the 98th percentile of patient treatment time is below 4 hours?" Finally, we would like to apply natural language techniques for Performance Trees in the context of important domains outside of modelling such as the specification of Service Level Agreements.

References

1. PIPE: Platform-Independent Petri net Editor – http://pipe2.sourceforge.net.
2. M. Ajmone-Marsan, G. Conte, and G. Balbo. A class of Generalised Stochastic Petri Nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems*, 2:93–122, 1984.
3. A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Verifying continuous-time Markov chains. In *Lecture Notes in Computer Science 1102: Computer-Aided Verification*, pages 269–276. Springer-Verlag, 1996.

4. A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Model checking continuous-time Markov chains. *ACM Transactions on Computational Logic*, 1(1):162–170, 2000.
5. P. Bonet, C.M. Llado, R. Puijaner, and W.J. Knottenbelt. PIPE v2.5: A Petri net tool for performance modelling. In *Proceedings of the 23rd Latin American Conference on Informatics (CLEI 2007)*, San Jose, Costa Rica, October 2007.
6. D.K. Brien, N.J. Dingle, W.J. Knottenbelt, H. Kulatunga, and T. Suto. Performance Trees: Implementation And Distributed Evaluation. In *Proc. 7th Intl. Workshop on Parallel and Distributed Methods in Verification (PDMC'08)*, Budapest, Hungary, March 2008. Elsevier.
7. N.J. Dingle. *Parallel Computation of Response Time Densities and Quantiles in Large Markov and Semi-Markov Models*. PhD thesis, Imperial College, London, United Kingdom, 2004.
8. S. Flake, W. Müller, and J. Ruf. Structured English for model checking specification. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, pages 99–108, Frankfurt, February 2000.
9. L. Grunske. Specification patterns for probabilistic quality properties. In *Proc. 30th International Conference on Software Engineering (ICSE'08)*, pages 31–40, Leipzig, Germany, 2008.
10. A. Holt and E. Klein. A semantically-derived subset of English for hardware verification. In *Proc. 37th Annual Meeting of the Association for Computational Linguistics*, pages 451–456, Maryland VA, USA, 1999.
11. W.J. Knottenbelt. Generalised Markovian analysis of timed transition systems. Master's thesis, University of Cape Town, Cape Town, South Africa, July 1996.
12. S. Konrad and B.H.C. Cheng. Real-time specification patterns. In *Proc. 27th International Conference on Software Engineering (ICSE'05)*, pages 372–381, St. Louis MO, USA, 2005.
13. R.L. Smith, G.S. Avrunin, L.A. Clarke, and L.J. Osterweil. PROPEL: An approach supporting property elucidation. In *Proc. 24th International Conference on Software Engineering (ICSE'02)*, pages 11–21, Orlando FL, USA, 2002.
14. T. Suto, J.T. Bradley, and W.J. Knottenbelt. Performance Trees: A New Approach to Quantitative Performance Specification. In *Proc. 14th IEEE/ACM Intl. Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS 2006)*, pages 303–313, Monterey, CA, USA, September 2006.
15. T. Suto, J.T. Bradley, and W.J. Knottenbelt. Performance trees: Expressiveness and quantitative semantics. In *Proceedings of the 4th International Conference on the Quantitave Evaluation of Systems (QEST'07)*, pages 41–50, Edinburgh, September 2007. IEEE Computer Society.