XML Query Evaluation via CTL Model Checking

Loredana Afanasiev

March 11, 2004

ILLC Master's thesis
Advisor: Massimo Franceschet

Abstract

The Extensible Markup Language (XML) was designed to describe the content of a document and its hierarchical structure. The XML Path language (XPath) is a language for selecting elements from XML documents. We propose and implement a *linear* embedding of the query evaluation problem for Core XPath, the navigational fragment of XPath, into the model checking problem for Computation Tree Logic (CTL). We extend the embedding to XCPath, an extension of Core XPath that is as expressive as first-order logic on sibling-ordered trees. This allows us to evaluate XCPath queries by exploiting a model checker for CTL. We report on experiments with the state-of-the-art model checker NuSMV, and compare our results with alternative academic XPath processors.

Acknowledgements

This thesis is the result of work with several people which I will mention below. I was just being in the middle of the process, trying to catch and to learn as much as possible, about myself, about my interests, about my capacities. What I have learned actually is a great deal about the people that I got to know. I learned how people can be generous and patient, professional and loving. I had a school of life more then a school of logic.

I consider this thesis to be the work of Massimo Franceschet, Maarten Marx and Balder ten Cate, as much as mine. More of mine was the pleasure of learning from them.

Maarten Marx's help cannot be described in words. He has many qualities which I respect. I will mention only one. He was able always to come up with just the right solution for my problems. He offered help firmly, quickly, the right one and in the right moment. I wish I could be the same in this respect.

Massimo Franceschet was very kind to me and brave, agreeing to be my adviser. He advised me wisely, caring more about my health than about deadlines.

Balder ten Cate didn't leave my side during the whole period. He helped me in every existing possible way. Balder brings up the best of me.

Maarten de Rijke and Dick de Jongh are the rare cases of the good people in the important administrative positions. They decided the course of things for me. Their kindness and goodness are legendary in ILLC. They gave me an ocean of help spread with patience.

Arjen de Vries made me a great favor by being in my commitee. His enthusiasm about science and about this project energized me. He is a bright source of inspiration.

Deep respect and warm feelings I have for Paul Vitanyi, who gave me incredible support and the most interesting hours full with discussions. His smiling eyes and almost child like interest and curiosity I'll never forget. I want to thank also Rudi Cilibrasi for his help.

I hope my road will cross the road of these people over and over again.

Humble admiration I have for Ineke ten Cate. She is an endless source of love and care.

Big gratitude goes to Nuffic and NWO for their financial support. In particular, I would like to thank Neda Wimmers for her help and patience with me.

At the end I engrave two names Ecaterina Ion and Vladimir Vasile Afanasiev. Everything I am and do is with the love of my parents.

Contents

Chapter 1

Aperitif

1.1 The Query Evaluation problem for XML

The Extensible Markup Language (XML) is a standard adopted by the World Wide Web Consortium (W3C) [16] that was designed to describe the content of a document, and its hierarchical structure. It is becoming a major standard for exchanging structured or semistructured data on the web [1].

XML query evaluation problem is the problem of selecting elements that meet particular requirements from an XML document. XML query evaluation is widely studied from the perspective of storing large databases as XML documents. As Buneman et al [8] observe, *"whether we shall store large XML document as databases (as opposed to using conventional databases and employing XML just for data exchange) depends on our ability to find specific XML storage models that support efficient querying of XML"*.

XML path language (XPath) [17] is a language proposed by the World Wide Web Consortium (W3C) for querying XML documents. XPath is already widely used as XML query language, and it there are several implementations of XPath retrieval systems. The XPath query evaluation problem is known to be solvable in polynomial time (polynomial in the combined size of the query and the document) [20]. Surprisingly, it was observed by the same authors that the existing XPath query evaluation processors, both the commercial and the academic onces, implement a naive, exponential-time algorithm. Since then, several new algorithms have been proposed. At present there are academic implementations of polynomial time algorithms for XPath query evaluation and linear time algorithms for smaller fragments of XPath, in particular Core XPath [20, 21]. Core XPath is simply the navigational part of XPath language.

In this thesis we focus on the query evaluation problem for Core XPath. We introduce and test a new method for solving this problem. Our algorithm also works for $\mathcal{X}$CPath, a recently introduced extension of Core XPath that is expressively complete for first-order logic [25].

1.2 The Model Checking problem for CTL

Model checking is the algorithmic verification that a given logical formula holds in a given structure. This concept is meaningful for most logics and classes of models, but it has been investigated most extensively for the case of *temporal logic*. During the last two decades, an active research field has emerged that focuses around the use of temporal logic model checking for the verification of software specifications. One temporal logic that has received much attention in this area is *Computation Tree Logic* (CTL) [15]. There are several implementations of efficient model checking algorithms for CTL, among which NuSMV [12].

While the computational complexity of the CTL model checking problem is very low (it can be performed in linear time), a lot of research has been directed at optimizing the existing model checking techniques. The reason is that in general, verification of computer systems requires that

formulas are checked on *very* large models, since the models essentially describe the entire state space of the program to be verified. It is therefore crucial to use an efficient representation of the model. Efficient in the sense that the representation is small, and in the sense that the model checking of temporal formulas can be performed efficiently given that the model is represented in this way. One technique that is often used to achieve this (and that is used among others by NuSMV) is *symbolic model checking*. Symbolic model checking algorithms avoid explicitly storing and manipulating the entire state space by using *Ordered Binary Decision Diagrams* (OBDDs). This model checking algorithm is called *symbolic* because it is based on manipulation of Boolean formulas.

1.3 Our proposal: Query Evaluation via Model Checking

The main idea that drives this thesis is to perform query evaluation via model checking: if we think of an XPath query as a modal formula, and of an XML document as a model, query evaluation is nothing more than checking which elements of the model satisfy the formula. This thesis is meant to test if this approach to query evaluation is feasible, both theoretically and in practice.

The idea of XML query evaluation via model checking is sensible for the following reason. One of the main concerns in the area of model checking is finding an efficient representation of models that admits efficient model checking. Various optimization techniques have been developed and implemented for exactly this purpose (in particular *symbolic model checking*). We might expect that these techniques work well also in the case of XML.

In Chapter 3, we establish a linear time reduction from the query evaluation problem of Core XPath to the model checking problem for CTL. In Chapter 4, we discuss an implementation of this translation, i.e., a Core XPath query evaluation system that works by translation to CTL. The system uses the CTL model checker NuSMV. Chapter 4 also contains a discussion of experimental results.

Our contributions can be summarized as follows.

- We present a linear time translation from the query evaluation problem of Core XPath to the model checking problem for CTL. This shows that query evaluation for Core XPath is in linear time. As a theoretical result, this was already known [20]. However, our translation can be used for practical implementation.

- We have implemented a Core XPath query evaluation system, XCheck, based on our translation. XCheck translates the XML query into CTL, and runs the CTL model checker NuSMV to evaluate it.

- We have performed experiments to analyze the practical efficiency of our proposal. Our first experimental results are not very positive. They show that there is a clear bottleneck in our approach, namely in representing the XML document symbolically, as input for the model checker. A more efficient symbolic representation of the XML document is needed in order to take full advantage of the model checking techniques.

- Our translation not only work for Core XPath, but also for a recently introduced extension of it, $\mathcal{X}$CPath.

  $\mathcal{X}$CPath has the property that it is expressively complete with respect to first-order logic. While our implementation only accepts Core XPath input, it is very easily extended to full $\mathcal{X}$CPath, thus leading to the first implementation of a query evaluation system for $\mathcal{X}$CPath.

1.4 Previous work

Since the mid-1990s there has been a lot of work on the interface of computational logic and semistructured data, making use of a wide variety of logical tools and techniques. For instance, [7]

uses simulations and morphisms, [2] concentrate on regular expressions, and [10] make the connection with description logic. Early work in the area was (necessarily) mostly concerned with proposals for data models and query languages for semistructured data. After the introduction of XML and XPath, this is where much of the research converged.

The relation between model checking and query processing has been extensively explored in the setting of *structured* data. We refer to [22] as a good entry-point to the area. The relation between model checking and query processing for *semistructured* data goes back at least to [3], where it was formulated in terms of suitable modal-like logics. Quintarelli [28] embeds a fragment of the graphical query language G-Log into CTL, and she sketches a mapping for subsets of other semistructured query languages, like Lorel, GraphLog and UnQL. De Alfaro [5] proposes the use of model checking for detecting errors in the structure and connectivity of web pages. Alechina et al. [4] discuss the use of Propositional Dynamic Logic (rather than CTL) to obtain decidability and complexity results for checking path constraints on semistructured data (rather than query evaluation). Calvanese et al. [10] use description logics for similar purposes. Finally, Miklau and Suciu [27] and Gottlob et al. [19] sketch an embedding of the forward looking fragment of XPath into CTL, but they do not test the practical effectiveness of this approach.

As noticed by Gottlob et al. [20], many available commercial engines implement XPath processing adopting a naive exponential-time strategy even though the query processing problem for XPath admits a polynomial-time algorithm, and it can be solved in linear time in case of Core XPath, the navigational fragment of XPath. A number of Core XPath processors have so far been implemented. In particular, in [20] the authors propose an algorithm that embeds a Core XPath query into an algebraic expression over sets of nodes of the tree representing an XML document, and that evaluates the algebraic expression in order to process the query. Later, Buneman et al. propose an algorithm for Core XPath processing on compressed XML files [8]. The core idea here is to compress the XML tree into a directed acyclic graph sharing common subtrees, and to evaluate the query directly on compressed XML documents. Moreover, Koch [23] embeds a query into a tree automaton and runs the resulting tree automaton in order to process the query. A different approach to process XPath on XML files is to embed XML documents into *relational* databases, to rewrite XPath queries as SQL ones, and to run an SQL engine to retrieve the answer set of the original XPath query (see, e.g., [31]). The advantage of such an approach is clear: the exploiting of existing relational database query processing technology, like index structures.

Chapter 2

The ingredients

2.1 Extensible Markup Language (XML)

The Extensible Markup Language (XML) is a standard adopted by the World Wide Web Consortium (W3C) [16] that was designed to describe the content of a document, and its hierarchical structure. It is becoming a major standard for exchanging structured or semistructured data on the web [1]. An example of XML document is shown in Figure 2.1.

The basic component in XML is the *element*, that is, a piece of text bounded by matching tags such as `<book>` and `</book>`. Inside an element we may have raw text or other elements. There are *composite* and *atomic* elements. Each atomic element consists of (1) a *tag*, (2) a possibly empty list of *attributes* of the form *name=value*, and (3) a possibly empty string of text called *Parsed Character Data* (PCDATA). Each composite element consists of a tag and an attribute list as above, and a list of *subelements*. For instance, consider the atomic element from Figure 2.1:

```
<title language="English"> Database Systems </title>
```

The tag of this element is `title`, the attribute list contains only the pair `language="English"`, and the PCDATA is `Database Systems`.

Next, consider the following composite element from Figure 2.1:

```
<book>
   <author> Cassio </author>
   <title> Data Production </title>
</book>
```

The tag is `book`, the attribute list is empty, and the list of subelements contains the two atomic elements with tags `author` and `title` respectively.

There exists always a main element, called the *root element*, in an XML document. The root element of the XML document in Figure 2.1 is the element with tag `biblio`. The attribute list of an element may contain a special attribute called *object identifier*, that can be used to identify the object uniquely. The object identifier is a special attribute with name `id`, the value of which must be uniquely identify the element.

An XML document may contain different elements with the same tag. However, attribute names have to be unique for the same element (i.e., the attribute list of an element cannot contain two different attributes with the same name). There is a document order between elements in an XML file, which is induced by the order in which the tags appear in the file. For instance, in Figure 2.1, `<author> Roux </author>` comes before `<author> Combalusier </author>`. We follow [8] in using the word *skeleton* to refer to what remains when all PCDATA is stripped off from an XML document.

The skeleton of an XML document can be represented as a finite sibling ordered node-labeled tree. Each element is represented as a node with all the subelements as its children. The root

```
<biblio>
  <book>
    <author> Roux </author>
    <author> Combalusier </author>
    <title language="English"> Database Systems </title>
    <date format = "YYYY"> 1999 </date>
  </book>
  <book>
    <author> Smith </author>
    <title> Database Systems </title>
    <date> 1999 </date>
  </book>
  <paper>
    <author> Cassio </author>
    <title> Data Production </title>
  </paper>
</biblio>
```

Figure 2.1: An XML document

element is the root of the tree, composite elements correspond to internal nodes, and atomic ones correspond to leaf nodes. The tree is sibling ordered in the sense that the children of a node, if any, are ordered (first child, second child, and so on) according to the document order. Finally, each node is labeled with its element tag, as well as with its attributes. Figure 2.2 contains the tree representation of the skeleton of the XML document in Figure 2.1. For simplicity, in this picture, the attribute information of the elements is not represented, and each node is labeled with a number to indicate the document order.

Throughout this thesis, we will only consider *skeletons* of XML documents. Moreover, we will ignore all attribute information, focusing only on the hierarchical structure of the document and the tags of the elements. The reason for this is that the XML query languages we will consider are not capable of describing more than just this part of the XML document.

We will now give a precise definition of the tree representation of XML documents that we will use in the next sections.

Definition 2.1.1 (XML Trees) *Let Σ be a set of labels corresponding to XML tags. An XML tree is a finite sibling ordered node-labeled tree $T = (N, R_{\downarrow}, R_{\rightarrow}, L)$, where N is the set of nodes, $R_{\downarrow} \subseteq N \times N$ is the set of edges of the tree, $R_{\rightarrow} \subseteq N \times N$ is a functional relation associating each node with its immediate right sibling (if any) and $L : \Sigma \rightarrow \wp(N)$ is a node-labeling function associating to each label a set of nodes.*

This definition is arguably too liberal: it does not prohibit cases in which an element satisfies two XML tags. None of the results in the rest of this thesis depend in any way on this simplification.

2.2 XML Path Languages

XML path language (XPath) [17] is a language proposed by the World Wide Web Consortium (W3C) for querying XML documents. Using XML queries, one can select elements from an XML document that meet particular requirements. XPath is already widely used as XML query language, and it there are several implementations of XPath retrieval systems.

XPath queries describe paths through an XML document. The queries are always of the form *locationstep/locationstep/. . . /locationstep* or */locationstep/locationstep/. . . /locationstep* (notice the intuitive similarity with Unix directory paths). In its unabbreviated form, each location

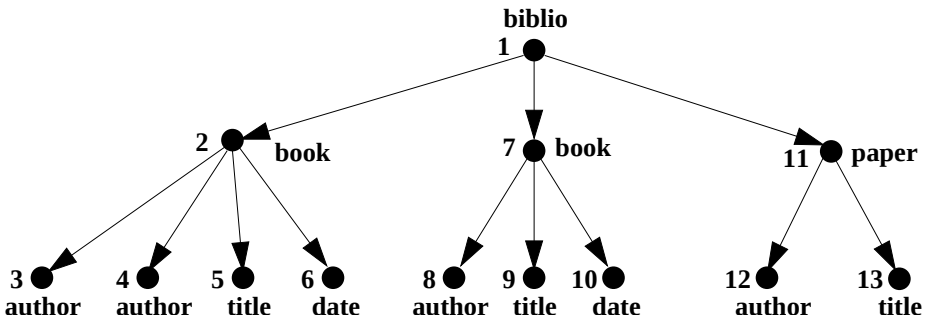


Figure 2.2: An XML tree

step is of the form $axis :: l$, where *axis* is an axis such as *child*, *parent* or *descendant*, and l is an XML tag. For example, `child::book` returns all subelements that have the XML tag `book`. In addition, predicates can be used to put additional requirements on the elements. For instance, `descendant::toy[attribute::color = "red"]` returns all subelements that have tag `toy` and that have attribute `color="red"`.

Being designed for practical use, XPath offers a range of numerical and string operations that can be used in formulating the queries. For example,

```
/child::biblio/child::book [attribute::date > 1998]
```

returns all books from the XML document in Figure 2.1 that have a `date` attribute with value higher than 1998. Similarly,

```
/child::biblio/child::book [string-length(child::title) > 3]/child::author
```

extracts form our example document the authors of all books that have a title consisting of more than three characters.

Rather than giving a formal definition of the full XPath language, we will proceed and consider two fragments of it, namely Core XPath and $\mathcal{X}$CPath.

2.2.1 Core XPath

Core XPath [20] is the clean logical core of XPath. The only objects that are manipulated by Core XPath are sets of nodes. No arithmetical or string operations are available. Hence, only the *skeleton* of the XML documents is considered in queries. The PCDATA and the attribute information of the elements is not manipulated.

The crucial notions of Core XPath language are axes, location steps and location paths. There are eleven *axes*: `self`, `child`, `parent`, `descendant`, `ancestor`, `descendant_or_self`, `ancestor_or_self`, `following_sibling`, `preceding_sibling`, `following`, and `preceding`. Each axis corresponds to a binary relation on the set of nodes of the tree. For instance, in the example of Figure 2.2, the axis `following` relates the node 7 with the nodes 11, 12, and 13, that is with the set of XML elements following the second book element in the XML document in Figure 2.1.

A *location path* is an arbitrary long sequence of location steps separated by the symbol /, where a *location step* has the form $axis :: l$ or $axis :: l[pred]$, where axis is one of the eleven axes, l an XML tag and *pred* a predicate. The location step $axis :: l$ is interpreted as a binary relation on the set of nodes of the tree that relates (n, m) if m is reachable from n through *axis* and the label of m is l. For example, the pair of nodes $(7, 13)$ belongs to the relation identified by `following::title`.

The location step $axis :: l[pred]$ introduces additional constraints on node selection. The location step $axis :: l[pred]$ is interpreted as a binary relation on the set of nodes of the tree that relates (n, m) if m is reachable from n through *axis*, the label of m is l and m satisfies the predicate *pred*. The predicate *pred* is a Boolean combination of locations paths. For instance, the pair of nodes $(1, 2)$ belongs to `child::book [child::author]`.

Finally, a location path *locationpath*/*locationstep* is interpreted as the concatenation of the relation for *locationpath* with the relation for *locationstep*. For example, the pair $(1, 3)$ belongs to `child::book/child::author`.

Location paths can be absolute or relative. Absolute location paths start with the symbol / and are evaluated from the root of the tree. Relative location paths miss the symbol / in the beginning and are evaluated at any node of the tree. A Core XPath query q is an absolute or relative location path and the result of the evaluation of q is the set of nodes that can be reached through the corresponding relation starting at the tree root or at any node of the tree, respectively.

We now we give the formal definition of the syntax of Core XPath [20].

Definition 2.2.1 (Core XPath Language) *Let Σ be a set of labels corresponding to the XML element tags. An* Core XPath *query is a formula generated by the first clause of the following inductive definition.*

$$\begin{array}{rcl} \text{cxp} & = & \text{locationpath} \mid /\text{locationpath} \\ \text{locationpath} & = & \text{locationstep}(/\text{locationstep})^* \\ \text{locationstep} & = & \chi :: l \mid \chi :: l[\text{pred}] \\ \text{pred} & = & \text{pred and pred} \mid \text{pred or pred} \mid \text{not pred} \mid \text{cxp} \\ \chi & = & \text{child} \mid \text{parent} \mid \text{descendant} \mid \text{ancestor} \mid \\ & & \text{descendant\_or\_self} \mid \text{ancestor\_or\_self} \mid \text{following\_sibling} \mid \\ & & \text{preceding\_sibling} \mid \text{following} \mid \text{preceding} \end{array}$$

where $l \in \Sigma \cup \{*\}$.

The label $*$ acts as a wildcard (i.e., denotes any possible XML element tag). Instead of stating the semantics for Core XPath directly, we will first discuss a more expressive variant of it, namely $\mathcal{X}$CPath [25], and we will show how Core XPath can be embedded into $\mathcal{X}$CPath. In this way, we implicitly give the semantics for Core XPath.

2.2.2 $\mathcal{X}$CPath

$\mathcal{X}$CPath is a more expressive variant of Core XPath that was defined and proved to be expressively complete with respect to first-order logic on XML trees in [25]. In the same paper the author proves that model checking for this language can be performed in linear time on both query and model size, by means of a translation to Propositional Dynamic Logic.

The syntax of $\mathcal{X}$CPath is as follows.

Definition 2.2.2 ($\mathcal{X}$CPath Language) *Let Σ be a set of labels corresponding to XML element tags. An $\mathcal{X}$*CPath *query is a formula generated by the first clause of the following inductive definition.*

$$\begin{array}{rcl} \textit{locationpath} & ::= & /\textit{locationstep} \mid \textit{locationstep} \mid \textit{locationpath}/\textit{locationstep} \\ \textit{locationstep} & ::= & \textit{axis} :: l \mid \textit{axis} :: l[\textit{pred}] \\ \textit{pred} & ::= & \textit{pred} \text{ and } \textit{pred} \mid \textit{pred} \text{ or } \textit{pred} \mid \text{ not } \textit{pred} \mid \textit{locationpath} \\ \textit{axis} & ::= & \epsilon \mid d \mid d^* \mid (d[\textit{pred}])^* \mid ([\textit{pred}]d)^* \end{array}$$

where $l \in \Sigma \cup \{*\}$ and $d \in \{\uparrow, \downarrow, \leftarrow, \rightarrow\}$.

Core XPath is contained in $\mathcal{X}$CPath, because the 11 axes of Core XPath can be defined in terms of those of $\mathcal{X}$CPath, cf. Table 2.1. On the other hand, $\mathcal{X}$CPath is more expressive then Core XPath [25].

For any binary relation R, let R^{-1} denote its inverse[1] and let R^* denote its transitive reflexive closure[2]. The semantics of an $\mathcal{X}$CPath query over an XML tree T is given by the following

[1] $R^{-1} = \{(m, n) \mid (n, m) \in R\}$.

[2] R^* is the smallest transitive reflexive relation that contains R.

Table 2.1: Traditional XPath axes vs. $\mathcal{X}$CPath axes

$$
\begin{array}{rcl}
\text{self} :: l[pred] & \equiv & \epsilon :: l[pred] \\
\text{child} :: l[pred] & \equiv & \downarrow :: l[pred] \\
\text{parent} :: l[pred] & \equiv & \uparrow :: l[pred] \\
\text{descendant} :: l[pred] & \equiv & \downarrow :: */ \downarrow^* :: l[pred] \\
\text{ancestor} :: l[pred] & \equiv & \uparrow :: */ \uparrow^* :: l[pred] \\
\text{descendant\_or\_self} :: l[pred] & \equiv & \downarrow^* :: l[pred] \\
\text{ancestor\_or\_self} :: l[pred] & \equiv & \uparrow^* :: l[pred] \\
\text{following\_sibling} :: l[pred] & \equiv & \rightarrow :: */ \rightarrow^* :: l[pred] \\
\text{preceding\_sibling} :: l[pred] & \equiv & \leftarrow :: */ \leftarrow^* :: l[pred] \\
\text{following} :: l[pred] & \equiv & \uparrow^* :: */ \rightarrow :: */ \rightarrow^* :: */ \downarrow^* :: l[pred] \\
\text{preceding} :: l[pred] & \equiv & \uparrow^* :: */ \leftarrow :: */ \leftarrow^* :: */ \downarrow^* :: l[pred]
\end{array}
$$

three simultaneously inductively defined functions: $\{\!|\cdot|\!\}_T$ that, given an $\mathcal{X}$CPath query, returns the corresponding binary relation on the set of nodes of T, $[\![\cdot]\!]_T$ that, given a predicate, returns a set of nodes of T that satisfy the predicate, and $\langle\!|\cdot|\!\rangle_T$ that, given an axis, returns the corresponding binary relation on the set of nodes of T.

Definition 2.2.3 (Semantics of $\mathcal{X}$CPath [25]) *Let $T = (N, R_\downarrow, R_\rightarrow, L)$ be an XML tree (cf. Definition 2.1.1). Let $R_\uparrow = (R_\downarrow)^{-1}$, $R_\leftarrow = (R_\rightarrow)^{-1}$ and let $L(*) = N$. Let root $\in N$ be the root of the tree T. We define the semantics of $\mathcal{X}$CPath as follows.*

$$
\begin{array}{rcl}
\{\!|\text{axis} :: l|\!\}_T & = & \{(n,m) \in \{\!|\text{axis}|\!\}_T \mid m \in L(l)\} \\
\{\!|\text{axis} :: l[\text{pred}]|\!\}_T & = & \{(n,m) \in \{\!|\text{axis}|\!\}_T \mid m \in L(l) \text{ and } m \in [\![pred]\!]_T\} \\
\{\!|/\text{locationstep}|\!\}_T & = & \{(n,m) \mid n \in N \text{ and } (\text{root}, m) \in \{\!|\text{locationstep}|\!\}_T\} \\
\{\!|\text{locationpath/locationstep}|\!\}_T & = & \{(n,k) \mid \exists m.(n,m) \in \{\!|\text{locationpath}|\!\}_T \text{ and } (m,k) \in \{\!|\text{locationstep}|\!\}_T\}
\end{array}
$$

$$
\begin{array}{rcl}
[\![\text{pred}_1 \text{ and } \text{pred}_2]\!]_T & = & [\![\text{pred}_1]\!]_T \cap [\![\text{pred}_2]\!]_T \\
[\![\text{pred}_1 \text{ or } \text{pred}_2]\!]_T & = & [\![\text{pred}_1]\!]_T \cup [\![\text{pred}_2]\!]_T \\
[\![\text{not pred}]\!]_T & = & N \setminus [\![\text{pred}]\!]_T \\
[\![\text{locationpath}]\!]_T & = & \{n \mid \exists m.(n,m) \in \{\!|\text{locationpath}|\!\}_T\}
\end{array}
$$

$$
\begin{array}{rcl}
\langle\!|\epsilon|\!\rangle_T & = & \{(n,n) \mid n \in N\} \\
\langle\!|d|\!\rangle_T & = & R_d \\
\langle\!|d^*|\!\rangle_T & = & R_d^* \\
\langle\!|(d[pred])^*|\!\rangle_T & = & \{(n,m) \in R_d \mid m \in [\![pred]\!]_T\}^* \\
\langle\!|([pred]d)^*|\!\rangle_T & = & \{(n,m) \in R_d \mid n \in [\![pred]\!]_T\}^*
\end{array}
$$

The *result* of evaluation of an $\mathcal{X}$CPath query q on a tree T is the set $\text{Result}(T,q) = \{m \mid \exists n.(n,m) \in \{\!|q|\!\}_T\}$. Notice that, given a query $q = /q'$, the result of the absolute query q and that of the relative one q' may differ.

2.3 Computation Tree Logic

Computation Tree Logic, or CTL for short, is a very expressive temporal logic, often used for proving the correctness of the computer systems via model checking [15]

Let Σ be a set of *propositional variables* whose elements are usually denoted by $l, a, b, c, \ldots$.

Definition 2.3.1 (CTL Language) *The* formulas *of* CTL *language are inductively defined by:*

$$\alpha \quad ::= \quad \text{true} \mid l \mid \alpha \wedge \alpha \mid \alpha \vee \alpha \mid \neg\alpha \mid \mathbf{EX}\alpha \mid \mathbf{EU}(\alpha,\alpha) \mid \mathbf{AU}(\alpha,\alpha)$$

Table 2.2: Intuitive readings of the temporal connectives of CTL

$\mathbf{EX}\alpha$:	on som**E** path, α is true in the ne**X**t state
$\mathbf{AX}\alpha$:	on **A**ll paths, α is true in the ne**X**t state
$\mathbf{EF}\alpha$:	on som**E** path, in some **F**uture state α is true
$\mathbf{AF}\alpha$:	on **A**ll paths, in some **F**uture state α is true
$\mathbf{EG}\alpha$:	on som**E** path, in all future states(**G**lobally) α is true
$\mathbf{AG}\alpha$:	on **A**ll paths, in all future states(**G**lobally) α is true
$\mathbf{EU}(\alpha_1, \alpha_2)$:	on som**E** path, α_1 is true **U**ntil α_2 is true
$\mathbf{AU}(\alpha_1, \alpha_2)$:	on **A**ll paths, α_1 is true **U**ntil α_2 is true

where $l \in \Sigma$.

In Definition 2.3.1, $\wedge$, $\vee$ and $\neg$ are standard logical connectives. The connectives **EX**, **EU**, **AU** are called *temporal connectives* and using them we can obtain five other temporal connectives: $\mathbf{AX}\alpha = \neg\mathbf{EX}\neg\alpha$, $\mathbf{EF}\alpha = \mathbf{EU}(\text{true}, \alpha)$, $\mathbf{AF}\alpha = \mathbf{AU}(\text{true}, \alpha)$, $\mathbf{EG}\alpha = \neg\mathbf{AF}\neg\alpha$, and $\mathbf{AG}\alpha = \neg\mathbf{EF}\neg\alpha$. The temporal connectives have the readings given in Table 2.2.

A CTL formula is interpreted over node-labeled, graphs $G = (N, E, L)$, where N is a set of nodes, $E \subseteq N \times N$ is a set of edges and $L : \Sigma \to \wp(N)$ is a node-labeling function. Hence, only the nodes are labeled, and not the edges. G must is a *total* with respect to E, meaning that for every $m \in N$ there is an $n \in N$ such that $(m, n) \in E$.

A *successor* of a node $n \in N$ is a node $m \in E$ with $(n, m) \in N$. A *path* in G is an infinite sequence $n_0, n_1, \ldots$, such that $(n_i, n_{i+1}) \in E$, for all n_i, n_{i+1} in the sequence.

We specify the semantics of CTL by means of the inductively defined function $[\![\cdot]\!]_G$ that, given a CTL-formula, returns the set of nodes from a given model G that satisfy the formula.

Definition 2.3.2 (Semantics of CTL [15]) *Let α be a* CTL*-formula and $G = (N, E, L)$ be a model for* CTL*. We define the semantics of* CTL *as follows:*

$$
\begin{aligned}
[\![\mathit{true}]\!]_G &= N \\
[\![l]\!]_G &= L(l),\ l \in \Sigma \\
[\![\alpha_1 \wedge \alpha_2]\!]_G &= [\![\alpha_1]\!]_G \cap [\![\alpha_2]\!]_G \\
[\![\alpha_1 \vee \alpha_2]\!]_G &= [\![\alpha_1]\!]_G \cup [\![\alpha_2]\!]_G \\
[\![\neg\alpha]\!]_G &= N \setminus [\![\alpha]\!]_G \\
[\![\mathbf{EX}\alpha]\!]_G &= \{n \in N \mid (n, m) \in E \text{ for some } m \in [\![\alpha]\!]_G\} \\
[\![\mathbf{EU}(\alpha_1, \alpha_2)]\!]_G &= \{n \in N \mid \text{ there is a path } \pi = n_0, n_1, \ldots \text{ such that } n_0 = n \text{ and} \\
&\quad \text{there is an } n_j \in \pi \text{ with } n_j \in [\![\alpha_2]\!]_G \text{ and } n_i \in [\![\alpha_1]\!]_G, \text{ for all } 0 \leq i < j\} \\
[\![\mathbf{AU}(\alpha_1, \alpha_2)]\!]_G &= \{n \in N \mid \text{ for all paths } \pi = n_0, n_1, \ldots \text{ such that } n_0 = n, \\
&\quad \text{there is an } n_j \in \pi \text{ with } n_j \in [\![\alpha_2]\!]_G \text{ and } n_i \in [\![\alpha_1]\!]_G, \text{ for all } 0 \leq i < j\}
\end{aligned}
$$

The following theorem by [14, 6] is discussed in [30].

Theorem 2.3.3 (CTL Model Checking) *The model checking problem for* CTL *can be solved in time* $O(|G| \cdot |\alpha|)$

Throughout this thesis, we will not make use of the **AU** modality. In particular, the translation from $\mathcal{X}$CPath to CTL we present in the next chapter never produces CTL formulas in which **AU** occurs. It turns out that the fragment of CTL without **AU** is contained in Propositional Dynamic Logic (PDL). Since PDL will not play any major role in this thesis, we will not define it here.

However, to see that CTL without the **AU** modalities is a fragment of PDL it suffices to note that $\mathbf{EX}\alpha$ and $\mathbf{EU}(\alpha_1, \alpha_2)$ can be written in PDL as $\langle a \rangle \alpha$ and $\langle (?\alpha_1; a)^* \rangle \alpha_2$ respectively, for some fixed action a.

2.3.1 Many dimensional CTL

We will also use a many dimensional variant of Computation Tree Logic, which we will refer to as CTL_Δ.

Definition 2.3.4 (The language of many dimensional CTL) *Let Σ be a set of node-labels, and let Δ be a finite set of edge-labels. The* formulas *of CTL_Δ are inductively defined by:*

$$\alpha \quad ::= \quad \text{true} \mid l \mid \alpha \wedge \alpha \mid \alpha \vee \alpha \mid \neg\alpha \mid \mathbf{EX}_{\mathrm{d}}\alpha \mid \mathbf{EU}_{\mathrm{d}}(\alpha, \alpha) \mid \mathbf{AU}_{\mathrm{d}}(\alpha, \alpha) \mid \mathbf{EU}'_{\mathrm{d}}(\alpha, \alpha)$$

where $l \in \Sigma$, $d \in \Delta$.

CTL_Δ formulas are interpreted over Δ-labeled graphs $G = (N, (E_d)_{d\in\Delta}, L)$, where N is a set of nodes, each $E_d \subseteq N \times N$ is a set of edges, $L : \Sigma \to \wp(N)$ is a node-labeling function. G must be total with respect to $\bigcup_{d\in\Delta} E_d$. The semantics is a straightforward generalization of that for one dimensional CTL. The $\mathbf{EU}'_{\mathbf{d}}$ is added to the language for technical reasons: it will allow us to formulate our translation from $\mathcal{X}\mathsf{CPath}$ to CTL more smoothly.

Generalizing the uni-relational case, a *path* in G is now an infinite sequence $n_0, n_1, \ldots$, such that $(n_i, n_{i+1}) \in \bigcup_{d\in\Delta} E_d$, for all n_i, n_{i+1} in the sequence.

Definition 2.3.5 (Semantics of Many Dimensional CTL) *Let α be a CTL_Δ formula and $G = (N, (E_d)_{d\in\Delta}, L)$ be a CTL_Δ model. We define the semantics of many dimensional* CTL *formulas as follows:*

$$\begin{aligned}
[\![true]\!]_G &= N \\
[\![l]\!]_G &= L(l) \\
[\![\alpha_1 \wedge \alpha_2]\!]_G &= [\![\alpha_1]\!]_G \cap [\![\alpha_2]\!]_G \\
[\![\alpha_1 \vee \alpha_2]\!]_G &= [\![\alpha_1]\!]_G \cup [\![\alpha_2]\!]_G \\
[\![\neg\alpha]\!]_G &= N \setminus [\![\alpha]\!]_G \\
[\![\mathbf{EX}_{\mathrm{d}}\alpha]\!]_G &= \{n \in N \mid (n, m) \in E_d \text{ for some } m \in [\![\alpha]\!]_G\} \\
[\![\mathbf{EU}_{\mathrm{d}}(\alpha_1, \alpha_2)]\!]_G &= \{n \in N \mid \text{ there is a path } \pi = n_0, n_1, \ldots \text{ such that } n_0 = n \text{ and} \\
&\quad \text{there is an } n_j \in \pi \text{ with } n_j \in [\![\alpha_2]\!]_G \text{ and } n_i \in [\![\alpha_1]\!]_G, (n_i, n_{i+1}) \in E_d \\
&\quad \text{for all } 0 \leq i < j\} \\
[\![\mathbf{EU}'_{\mathrm{d}}(\alpha_1, \alpha_2)]\!]_G &= \{n \in N \mid \text{ there is a path } \pi = n_0, n_1, \ldots \text{ such that } n_0 = n \text{ and} \\
&\quad \text{there is an } n_j \in \pi \text{ with } n_j \in [\![\alpha_2]\!]_G \text{ and } n_i \in [\![\alpha_1]\!]_G, (n_i, n_{i+1}) \in E_d \\
&\quad \text{for all } 0 \leq i < j\} \\
[\![\mathbf{AU}_{\mathrm{d}}(\alpha_1, \alpha_2)]\!]_G &= \{n \in N \mid \text{ for all paths } \pi = n_0, n_1, \ldots \text{ such that } n_0 = n, \\
&\quad \text{there is an } n_j \in \pi \text{ with } n_j \in [\![\alpha_2]\!]_G \text{ and } n_i \in [\![\alpha_1]\!]_G, (n_i, n_{i+1}) \in E_d \\
&\quad \text{for all } 0 \leq i < j\}
\end{aligned}$$

Note that we used the same function $[\![\cdot]\!]_G$ for the semantics of plain and many dimensional CTL. We assume that it will always be clear from the context which language we mean.

In the next chapter we will use CTL_Δ as an intermediate step in translating $\mathcal{X}\mathsf{CPath}$ into CTL. We will prove that $\mathcal{X}\mathsf{CPath}$ query evaluation problem can be reduced in linear time to CTL_Δ with four dimensions and for finite Δ, CTL_Δ model checking can be reduced in linear time to model checking for CTL. Therefore, Theorem 2.3.3 also holds to CTL_Δ (for finite Δ) and from this it follows that $\mathcal{X}\mathsf{CPath}$ query evaluation can be solved via our translation in linear time.

Chapter 3

The recipe

3.1 Embedding $\mathcal{X}$CPath into CTL

In this section, we give a linear time reduction from $\mathcal{X}$CPath query evaluation (hence from Core XPath) to CTL model checking. Using Theorem 2.3.3, we conclude that $\mathcal{X}$CPath query evaluation can be performed in linear time. As a theoretical result, this was already known [25]. However, our reduction to CTL is suitable for practical implementation.

The reduction proceeds in two steps. First, we linearly reduce $\mathcal{X}$CPath query evaluation to model checking for many dimensional CTL with four dimensions. Next, we show that many dimensional CTL can be linearly reduced to CTL.

3.1.1 Embedding $\mathcal{X}$CPath into many dimensional CTL

In this section, we translate the models and queries for $\mathcal{X}$CPath into models and formulas for many dimensional CTL. We reduce $\mathcal{X}$CPath query evaluation to model checking for many dimensional CTL. First, we convert an XML tree model into a many dimensional CTL model.

Definition 3.1.1 (From XML trees to CTL$_{\{\uparrow,\downarrow,\leftarrow,\rightarrow\}}$ models) *Let $T = (N, R_\downarrow, R_\rightarrow, L)$, with $L : \Sigma \to \wp(N)$ be an XML tree (cf. Definition 2.1.1). We define the translation of T to be the labeled graph $\mu_1(T) = (N, (E_d)_{d\in\{\downarrow,\uparrow,\leftarrow,\rightarrow\}}, L)$, with $E_\downarrow = R_\downarrow$, $E_\rightarrow = R_\rightarrow$, $E_\uparrow = (R_\downarrow)^{-1}$, $E_\leftarrow = (R_\rightarrow)^{-1}$*

Remark 3.1.2 The obtained model $\mu_1(T)$ is in almost all cases total with respect to $\bigcup_{d\in\{\downarrow,\uparrow,\leftarrow,\rightarrow\}} E_d$, thus we are allowed to talk about paths in the model. The only exception is the model $\mu_1(T)$, where T is an XML tree with a single node. In this thesis we will not consider this trivial case.

Notice that the size of $\mu_1(T)$ is linear in the size of T, and that $\mu_1(T)$ can be obtained from T in linear time.

Given an axis $axis$, let $axis^{-1}$ be its inverse, i.e. $\epsilon^{-1} = \epsilon$, $(\leftarrow)^{-1} = \rightarrow$, ... and $(d^*)^{-1} = (d^{-1})^*$. Finally, for $l \in \Sigma \cup \{*\}$, let $l' = true$ in case $l = *$ and $l' = l$ otherwise.

Definition 3.1.3 (From $\mathcal{X}$CPath queries to CTL$_{\{\uparrow,\downarrow,\leftarrow,\rightarrow\}}$ formulas) *Given a $\mathcal{X}$CPath query q, we define its translation into many dimensional CTL formula $\tau_1(q)$ as in Table 3.1, where* root $= \neg\mathbf{EX}_\uparrow$true.

One can easily see that the cases we consider for translating a predicate, although they do not follow the recursive definition of a location path, are exhaustive and deterministic. Also notice that the length of $\tau_1(q)$ is linear in the length of q, and that $\tau_1(q)$ can be obtained from q in linear time.

Table 3.1: Translation from $\mathcal{X}$CPath into $\mathsf{CTL}_{\{\uparrow,\downarrow,\leftarrow,\rightarrow\}}$

$$
\begin{array}{rcl}
\tau_1(axis :: l) & = & l' \wedge \langle axis^{-1}\rangle true \\
\tau_1(axis :: l[pred]) & = & l' \wedge \langle axis^{-1}\rangle true \wedge \omega_1(pred) \\
 & & \\
\tau_1(/axis :: l) & = & l' \wedge \langle axis^{-1}\rangle \mathrm{root} \\
\tau_1(/axis :: l[pred]) & = & l' \wedge \langle axis^{-1}\rangle \mathrm{root} \wedge \omega_1(pred) \\
 & & \\
\tau_1(locationpath/axis :: l) & = & l' \wedge \langle axis^{-1}\rangle \tau_1(locationpath) \\
\tau_1(locationpath/axis :: l[pred]) & = & l' \wedge \langle axis^{-1}\rangle \tau_1(locationpath) \wedge \omega_1(pred)
\end{array}
$$

$$
\begin{array}{rcl}
\omega_1(pred_1 \text{ and } pred_2) & = & \omega_1(pred_1) \wedge \omega_1(pred_2) \\
\omega_1(pred_1 \text{ or } pred_2) & = & \omega_1(pred_1) \vee \omega_1(pred_2) \\
\omega_1(\text{ not } pred) & = & \neg\omega_1(pred) \\
 & & \\
\omega_1(axis :: l) & = & \langle axis\rangle l' \\
\omega_1(axis :: l[pred]) & = & \langle axis\rangle (l' \wedge \omega_1(pred)) \\
 & & \\
\omega_1(/locationpath) & = & \mathbf{EF}_{\uparrow}(\mathrm{root} \wedge \omega_1(locationpath)) \\
\omega_1(axis :: l/locationpath) & = & \langle axis\rangle (l' \wedge \omega_1(locationpath)) \\
\omega_1(axis :: l[pred]/locationpath) & = & \langle axis\rangle (l' \wedge \omega_1(locationpath) \wedge \omega_1(pred))
\end{array}
$$

$$
\begin{array}{rcl}
\langle\epsilon\rangle\alpha & = & \alpha \\
\langle d\rangle\alpha & = & \mathbf{EX}_d\alpha \\
\langle d^*\rangle\alpha & = & \mathbf{EU}_d(true, \alpha) = \mathbf{EF}_d\alpha \\
\langle (d[pred])^*\rangle\alpha & = & \mathbf{EU}'_{\mathrm{d}}(\omega_1(pred), \alpha) \\
\langle ([pred]d)^*\rangle\alpha & = & \mathbf{EU}_{\mathrm{d}}(\omega_1(pred), \alpha)
\end{array}
$$

Intuitively, one can view the axis in a location step of a query as a CTL modality "looking in opposite direction", while the axis in a location step of a predicate can be seen as an "identical direction" CTL modality. The reason behind this is that the CTL translation of a query characterizes the nodes that lie on the *end* of a path satisfying the query, whereas the CTL translation of a predicate characterizes the nodes that lie on the *start* of such a path. The length of a query is nothing else then the depth of a the coresponding CTL formula.

Example 3.1.4 Consider the absolute XPath query:

```
q = /child::book [child::author [following_sibling::author]]/child::title
```

Its equivalent in our notation is:

$$\texttt{q = /}\downarrow\texttt{::book [}\downarrow\texttt{::author [}\rightarrow\texttt{::}*\texttt{ /}\rightarrow^*\texttt{::author]]/}\downarrow\texttt{::title}$$

It selects the title node of all the books with at least two authors. In the tree in Figure 2.2, the query q retrieves the node 2. Its translation $\tau_1(q)$ is the following many dimensional CTL formula:

$$\text{title} \wedge \mathbf{EX}_{\uparrow}(\text{book} \wedge \mathbf{EX}_{\uparrow}\text{root} \wedge \mathbf{EX}_{\downarrow}(\text{author} \wedge \mathbf{EX}_{\rightarrow}(\text{true} \wedge \mathbf{EF}_{\rightarrow}\text{author}))$$

which is equivalent to:

$$\text{title} \wedge \mathbf{EX}_{\uparrow}(\text{book} \wedge \mathbf{EX}_{\uparrow}\text{root} \wedge \mathbf{EX}_{\downarrow}(\text{author} \wedge \mathbf{EX}_{\rightarrow}\mathbf{EF}_{\rightarrow}\text{author})$$

Theorem 3.1.5 (Correctness) *Let T be a XML tree with more than one node, q an $\mathcal{X}$CPath query, pred an $\mathcal{X}$CPath predicate and α a many dimensional CTL formula. Then,*

1. $\text{Result}(T,q) = [\![\tau_1(q)]\!]_{\mu_1(T)}$
2. $[\![pred]\!]_T = [\![\omega_1(pred)]\!]_{\mu_1(T)}$
3. $[\![\langle axis\rangle\alpha]\!]_{\mu_1(T)} = \{m \mid \text{there is an } n \in [\![\alpha]\!]_{\mu_1(T)} \text{ such that } (m,n) \in R_{axis}\}$.

Proof. *By simultaneous induction on the length of q, pred and axis. Let $T = (N, R_\downarrow, R_\rightarrow, L_T)$, where $L_T : N \to \Sigma$, and $\mu_1(T) = (N, (E_d)_{d\in\{\downarrow,\uparrow,\leftarrow,\rightarrow\}}, L)$, where $L : \Sigma \to \wp(N)$. For convenience, let $L(*) = N$.*

Consider q of the form $/axis :: l[pred]$. The cases where $q = axis :: l[pred]$, $q = /axis :: l$, $q = axis :: l$ are similar.

$$\begin{array}{ll} & m \in Result(T,q) \\ \textit{iff} & (root,m) \in R_{axis} \text{ and } m \in L(l) \text{ and } m \in [\![pred]\!]_T \\ \textit{iff}_{(IHP)} & (root,m) \in R_{axis} \text{ and } m \in [\![l']\!]_{\mu_1(T)} \text{ and } m \in [\![\omega_1(pred)]\!]_{\mu_1(T)} \\ \textit{iff} & (m,root) \in R_{axis^{-1}} \text{ and } m \in [\![l']\!]_{\mu_1(T)} \text{ and } m \in [\![\omega_1(pred)]\!]_{\mu_1(T)} \\ \textit{iff}_{(IHP)} & m \in [\![l' \wedge \langle axis^{-1}\rangle\text{root} \wedge \omega_1(pred)]\!]_{\mu_1(T)} \\ \textit{iff} & m \in [\![\tau_1(q)]\!]_{\mu_1(T)} \end{array}$$

Consider the case where q is of the form $locationpath/axis :: l[pred]$. The case where $q = locationpath/axis :: l$ is similar.

$$\begin{array}{ll} & m \in Result(T,q) \\ \textit{iff} & \exists n.n \in Result(T, locationpath) \text{ and } (n,m) \in \{\!|axis :: l[pred]|\!\}_T \\ \textit{iff} & \exists n.n \in Result(T, locationpath) \text{ and } (n,m) \in R_{axis} \text{ and } m \in L(l) \\ & \text{and } m \in [\![pred]\!]_T \\ \textit{iff}_{(IHP)} & \exists n.n \in [\![\tau_1(locationpath)]\!]_{\mu_1(T)} \text{ and } (n,m) \in R_{axis} \text{ and } m \in [\![l']\!]_{\mu_1(T)} \text{ and} \\ & m \in [\![\omega_1(pred)]\!]_{\mu_1(T)} \\ \textit{iff} & \exists n.n \in [\![\tau_1(locationpath)]\!]_{\mu_1(T)} \text{ and } (m,n) \in R_{axis^{-1}} \text{ and } m \in [\![l']\!]_{\mu_1(T)} \text{ and} \\ & m \in [\![\omega_1(pred)]\!]_{\mu_1(T)} \\ \textit{iff}_{(IHP)} & m \in [\![l' \text{ and } \langle axis^{-1}\rangle\tau_1(locationpath) \text{ and } \omega_1(pred)]\!]_{\mu_1(T)} \\ \textit{iff} & m \in [\![\tau_1(q)]\!]_{\mu_1(T)} \end{array}$$

We now proceed with part 2. of the theorem. We will skip the Boolean cases, which are trivial. Consider the case where pred is of the form axis :: l[pred′]. The proof for the case pred = axis :: l is similar.

$$
\begin{array}{ll}
 & n \in [\![pred]\!]_T \\
\textit{iff} & \exists m.(n,m) \in \{\!|axis :: l[pred']|\!\}_T \\
\textit{iff} & \exists m.(n,m) \in R_{axis} \textit{ and } m \in L(l) \textit{ and } m \in [\![pred']\!]_T \\
\textit{iff } (IHP) & \exists m.(n,m) \in R_{axis} \textit{ and } m \in [\![l']\!]_{\mu_1(T)} \textit{ and } m \in [\![\omega_1(pred')]\!]_{\mu_1(T)} \\
\textit{iff } (IHP) & m \in [\![\langle axis\rangle(l' \textit{ and } \omega_1(pred))]\!]_{\mu_1(T)} \\
\textit{iff} & m \in [\![\omega_1(pred)]\!]_{\mu_1(T)}
\end{array}
$$

Let pred be of the form axis :: l[pred′]/locationpath. The pred = axis :: l/locationpath case is similar.

$$
\begin{array}{ll}
 & n \in [\![pred]\!]_T \\
\textit{iff} & \exists m.(n,m) \in \{\!|axis :: l[pred']|\!\}_T \textit{ and } m \in [\![locationpath]\!]_T \\
\textit{iff} & \exists m.(n,m) \in R_{axis} \textit{ and } m \in L(l) \textit{ and } m \in [\![pred']\!]_T \textit{ and } m \in [\![locationpath]\!]_T \\
\textit{iff } (IHP) & \exists m.(n,m) \in R_{axis} \textit{ and } m \in [\![l']\!]_{\mu_1(T)} \textit{ and } m \in [\![\omega_1(pred')]\!]_{\mu_1(T)} \textit{ and} \\
 & m \in [\![\omega_1(locationpath)]\!]_{\mu_1(T)} \\
\textit{iff } (IHP) & n \in [\![\langle axis\rangle(l' \wedge \omega_1(pred') \wedge \omega_1(locationpath))]\!]_{\mu_1(T)} \\
\textit{iff} & n \in [\![\omega_1(pred))]\!]_{\mu_1(T)}
\end{array}
$$

Let pred be of the form /locationpath.

$$
\begin{array}{ll}
 & n \in [\![pred]\!]_T \\
\textit{iff} & root \in [\![locationpath]\!]_T \\
\textit{iff } (IHP) & root \in [\![\omega_1(locationpath)]\!]_{\mu_1(T)} \\
\textit{iff} & n \in [\![\mathbf{EF}_{\uparrow}(\neg\mathbf{EX}_{\uparrow}\mathrm{true} \wedge \omega_1(locationpath))]\!]_{\mu_1(T)} \\
\textit{iff} & n \in [\![\omega_1(pred))]\!]_{\mu_1(T)}
\end{array}
$$

Finally, we proceed with part 3. of the theorem. Let axis be of the form ([pred]d). The other cases are similar.*

$$
\begin{array}{ll}
 & m \in [\![\langle axis\rangle\alpha]\!]_{\mu_1(T)} \\
\textit{iff} & m \in [\![\mathbf{EU_d}(\omega_1(pred), \alpha)]\!] \\
\textit{iff} & \textit{there is a path } \pi = n_0, n_1, \ldots \textit{ such that } n_0 = m, \textit{ and there is an } n_j \in \pi \textit{ with} \\
 & n_j \in [\![\alpha]\!]_{\mu_1(T)} \textit{ and } n_i \in [\![\omega_1(pred)]\!]_{\mu_1(T)}, (n_i, n_{i+1}) \in E_d \textit{ for all } 0 \leq i < j \\
\textit{iff } (IHP) & \textit{there is a path } \pi = n_0, n_1, \ldots \textit{ such that } n_0 = m \textit{ and there is an } n_j \in \pi \textit{ with} \\
 & n_j \in [\![\alpha]\!]_{\mu_1(T)} \textit{ and } n_i \in [\![pred]\!]_T, (n_i, n_{i+1}) \in E_d \textit{ for all } 0 \leq i < j \\
\textit{iff} & \textit{there is a node } n_j \in [\![\alpha]\!]_{\mu_1(T)} \textit{ such that } (m,n) \in R_{axis}
\end{array}
$$

QED

3.1.2 Embedding many dimensional CTL into one dimensional CTL

We will encode CTL_Δ into plain, one dimensional CTL, for finite Δ. First, we transform a CTL_Δ model into a simple, one dimensional CTL model.

The idea of the translation is based on the following intuition. For every node n and direction $d \in \Delta$, we create a new node (n, d). Next, we replace every transition of the form $n \xrightarrow{d} m$ by $(n, d) \to (m, d)$. In this way, replace the edge-labels by node-labels.

Definition 3.1.6 (From CTL_Δ models to CTL models) *Let $G = (N, (E_d)_{d\in\Delta}, L)$ be a CTL_Δ model, with $L : \Sigma \to \wp(N)$. We rewrite G into a node-labeled graph $\mu_2(G) = (N', E', L')$, with $L' : \Sigma \cup \Delta \cup \{a\} \to \wp(N')$, where a is a new label different from all $d \in \Delta$ and $l \in \Sigma$, such that:*

- $N' = N \times \Delta$;
- $E' = \{((u,a),(u,d)), ((u,d),(u,a)) \mid u \in N, d \in \Delta\} \cup \{((u,d),(v,d)) \mid (u,v) \in E_d, d \in \Delta\}$;
- $L'(l) = \{(u,a) \mid u \in L(l)\}$, *for* $l \in \Sigma$
  $L'(d) = \{(u,d) \mid u \in N\}$, *for* $d \in \Delta$
  $L'(a) = \{(u,a) \mid u \in N\}$

Notice that, for a fixed, finite Δ, the size of $\mu_2(G)$ is linear in the size of G, and that $\mu_2(G)$ can be obtained from G in linear time.

Definition 3.1.7 (From CTL_Δ formulas to CTL formulas) *The translation τ_2 from CTL_Δ to CTL is given as follows, where α, α_1 and α_2 are CTL_Δ formulas.*

$$\begin{array}{rcl}
\tau_2(true) & = & true \\
\tau_2(l) & = & l \\
\tau_2(\alpha_1 \wedge \alpha_2) & = & \tau_2(\alpha_1) \wedge \tau_2(\alpha_2) \\
\tau_2(\alpha_1 \vee \alpha_2) & = & \tau_2(\alpha_1) \vee \tau_2(\alpha_2) \\
\tau_2(\neg\alpha_1) & = & \neg\tau_2(\alpha_1) \\
\tau_2(\mathbf{EX}_d\alpha_1) & = & \mathbf{EX}(d \wedge \mathbf{EX}(d \wedge \mathbf{EX}(a \wedge \tau_2(\alpha_1)))) \\
\tau_2(\mathbf{EU}_d(\alpha_1,\alpha_2)) & = & \mathbf{EXEU}(d \wedge \mathbf{EX}(a \wedge \tau_2(\alpha_1)), d \wedge \mathbf{EX}(a \wedge \tau_2(\alpha_2))) \\
\tau_2(\mathbf{AU}_d(\alpha_1,\alpha_2)) & = & \mathbf{EXAU}(d \wedge \mathbf{EX}(a \wedge \tau_2(\alpha_1)), d \wedge \mathbf{EX}(a \wedge \tau_2(\alpha_2))) \\
\tau_2(\mathbf{EU'}_{\mathrm{d}}(\alpha_1,\alpha_2)) & = & \mathbf{EX}(d \wedge \mathbf{EXEU}(d \wedge \mathbf{EX}(a \wedge \tau_2(\alpha_1)), a \wedge \tau_2(\alpha_2)))
\end{array}$$

Again, notice that the length of $\tau_2(\alpha)$ is linear in the length of α, and that $\tau_2(\alpha)$ can be obtained from α in linear time and the operator depth can grow at most 4 times bigger. Given a set S of nodes of $\mu_2(G)$, we define $\Downarrow_a(S) = \{n \mid (n,a) \in S\}$.

Theorem 3.1.8 (Correctness) *Let $G = (N, (E_d)_{d\in\Delta}, L)$ be a CTL_Δ model and α be a CTL_Δ formula. Then,*

$$[\![\alpha]\!]_G = \Downarrow_a [\![\tau_2(\alpha)]\!]_{\mu_2(G)}$$

Proof. *Let $\mu_2(G) = (N', E, L')$. To prove the theorem, it suffices to prove the following equivalent statement:*

$$n \in [\![\alpha]\!]_G \quad \textit{iff} \quad (n,a) \in [\![\tau_2(\alpha)]\!]_{\mu_2(G)}$$

The proof goes by induction on the structure of α.

- *The* atomic cases: *Let $\alpha = true$. We have that $n \in [\![true]\!]_G = N$* iff *$(n,a) \in [\![true]\!]_{\mu_2(G)} = N'$. Let $\alpha = l$. We have that $n \in [\![l]\!]_G = L(l)$* iff *by the definition of $\mu_2(G)$, $(n,a) \in [\![l]\!]_{\mu_2(G)} = L'(l)$.*
- *The* Boolean cases *are simple.*
- *Let $\alpha = \mathbf{EX}_d\alpha'$. We have that $n \in [\![\mathbf{EX}_d\alpha']\!]_G$* iff *there is an m such that $(n,m) \in E_d$ and $m \in [\![\alpha']\!]_G$. By IHP and by definition of $\mu_2(G)$, the latter holds* iff *there is a node (m,a) in $\mu_2(G)$ such that $((n,a),(n,d))$, $((n,d),(m,d))$, $((m,d),(m,a))$ are in E and $(m,a) \in [\![\tau_2(\alpha')]\!]_{\mu_2(G)}$, which holds* iff *$(n,a) \in [\![\mathbf{EX}(d \wedge \mathbf{EX}(d \wedge \mathbf{EX}(a \wedge \tau_2(\alpha'))))]\!]_{\mu_2(G)} = [\![\tau_2(\alpha')]\!]_{\mu_2(G)}$.*
- *Let $\alpha = \mathbf{EU}_d(\alpha_1,\alpha_2)$. The case for α of the form $\mathbf{AU}_{\mathrm{d}}(\alpha_1,\alpha_2)$ is similar. We have that $n \in [\![\alpha]\!]_G$* iff *for some path in G, $n_0, n_1, n_2, \ldots$, with $n_0 = n$, there is some n_j on the path, such that $n_j \in [\![\alpha_2]\!]_G$ and $n_i \in [\![\alpha_1]\!]_G$, $(n_i, n_{i+1}) \in E_d$ for all $i < j$. By definition of $\mu_2(G)$ and by IHP, the latter holds* iff *there is a corresponding path $(n_0,a),(n_0,d),(n_1,d),(n_1,d),(n_2,d),\cdots,(n_j,d),(n_j,a)$ in $\mu_2(G)$ with $n_0 = n$, such that we have $(n_j,a) \in [\![\tau_2(\alpha_2)]\!]_{\mu_2(G)}$ and $(n_i,a) \in [\![\ \tau_2(\alpha_1)]\!]_{\mu_2(G)}$ for all $0 \le i < j$. This holds* iff *$(n,a) \in [\![\mathbf{EXEU}(d \wedge \mathbf{EX}(a \wedge \tau_2(\alpha_1)), d \wedge \mathbf{EX}(a \wedge \tau_2(\alpha_2)))]\!]_{\mu_2(G)} = [\![\tau_2(\alpha)]\!]_{\mu_2(G)}$.*
- *Let α be of the form $\mathbf{EU'}_{\mathrm{d}}(\alpha_1,\alpha_2)$. We have that $n \in [\![\alpha]\!]_G$* iff *for some path in G, $n_0, n_1, n_2, \ldots$, with $n_0 = n$, there is some n_j on the path, such that $n_j \in [\![\alpha_2]\!]_G$ and $n_i \in [\![\alpha_1]\!]_G$, $(n_i, n_{i+1}) \in E_d$ for all $1 \le i \le j$. By definition of $\mu_2(G)$ and by IHP, the latter holds* iff *there is a corresponding path $(n_0,a),(n_0,d),(n_1,d),(n_2,d),\cdots,(n_j,d),(n_j,a)$ in $\mu_2(G)$ with $n_0 = n$, such that we have $(n_j,a) \in [\![\tau_2'(\alpha_2)]\!]_{\mu_2(G)}$ and $(n_i,a) \in [\![\ \tau_2'(\alpha_1)]\!]_{\mu_2(G)}$ for all $1 \le i \le j$. The latter holds* iff *there is a path $(n_0,a),(n_0,d),(n_1,d),(n_2,d),\cdots,(n_j,d),(n_j,a)$ in $\mu_2(G)$ with $n_0 = n$, such that we have $(n_j,d) \in [\![d \wedge \mathbf{EX}(a \wedge \tau_2'(\alpha_2))]\!]_{\mu_2(G)}$ and $(n_i,a) \in [\![\ a \wedge \tau_2'(\alpha_1)]\!]_{\mu_2(G)}$ for all $1 \le i \le j$. This holds* iff *$(n,a) \in [\![\mathbf{EX}(d \wedge \mathbf{EXEU}(d \wedge \mathbf{EX}(\tau_2'(\alpha_1)), a \wedge \tau_2'(\alpha_2)))]\!]_{\mu_2(G)} = [\![\tau_2'(\alpha)]\!]_{\mu_2(G)}$.*

QED

Corollary 3.1.9 *For finite Δ, there is a linear time (both on query size and model size) reduction from CTL_Δ model checking to CTL model checking.*

Theorem 3.1.10 (Main) *Let T be a XML tree and q be a $\mathcal{X}\mathsf{CPath}$ query. Then,*

$$\mathrm{Result}(T, q) = \Downarrow_a [\![\tau_2(\tau_1(q))]\!]_{\mu_2(\mu_1(T))}$$

Proof. *Follows from Theorem 3.1.5 and Theorem 3.1.8* QED

Corollary 3.1.11 *There is a linear time (both on query size and model size) reduction from $\mathcal{X}\mathsf{CPath}$ query evaluation to CTL model checking. Thus, by Theorem 2.3.3 it follows that $\mathcal{X}\mathsf{CPath}$ query evaluation problem is solved in linear time.*

A worked-out example of the document translation can be found in Section 4.1.1.

Chapter 4

Cooking and Tasting

One advantage of our approach to query evaluation is that it is easily implemented. By using an existing CTL model checker, we reduce our task to implementing the query and document translations presented in Chapter 3 and the translation from the model checker output back to the world of XML.

We have indeed followed this approach, and the resulting implementation is called XCheck. We performed several experiments to analyze the behavior of XCheck, and to compare it to two other Core XPath query evaluation systems. The present chapter reports on XCheck, its implementation and on the obtained experimental results.

First, we will discuss the model checker that we used, NuSMV. We motivate our choice, discuss the features of NuSMV and describe its input requirements. We will also report on a small modification that we needed to make in order to use NuSMV for our purposes. Next, we provide a detailed description of our implementation, including the source code, which is written in Perl. Finally, we discuss the experiments that we performed and their outcome.

4.1 NuSMV model checker

NuSMV [13] is a state-of-the-art model checker for CTL and several other temporal logics. It is generally considered to be among the best model checkers currently available. It performs *symbolic* model checking [26]. Recall from Chapter 1 that symbolic model checkers cleverly avoid the need to represent the entire state space of the program by the use of ordered binary decision diagrams [15]. Moreover, NuSMV incorporates several model partitioning methods [9]. For programs with a large state space, this gives rise to a considerable improvement in the efficiency of the model checker.

Apart from this, NuSMV is open source, modularly structured and well documented, all of which make it attractive for our purposes. NUSMV is available at `http://nusmv.irst.itc.it/`. Its documentation can be found on the same website [11].

In spite of all its virtues, NuSMV lacks one feature that is necessary in order for us to be able to use it. Being designed for the verification of computer software, NuSMV only allows for *local model checking*, i.e., given a formula, a model and a state in that model, test whether the formula is satisfied or not. For our purposes, we need to solve a slightly more general task: given a formula and a model, list the states that satisfy the formula. Luckily, this functionality is easily obtained by making a small modification to the source code of NuSMV.

In the remainder of this section, we discuss NuSMV's input language, in which the model is specified, and the modification that was made in order allow for global model checking.

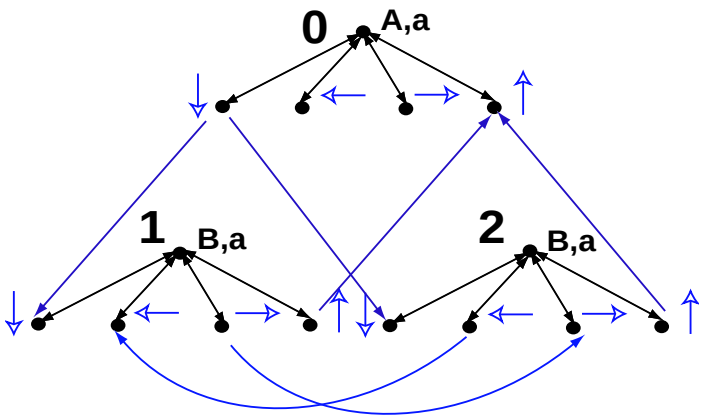


Figure 4.1: The CTL model

4.1.1 Model specifications

Since NuSMV is a symbolic model checker, it takes as input not a model and a formula, but a program specification and a formula. By taking as input a program specification, NuSMV avoids representing the full state space (i.e., the full model). Rather, it uses the program to construct a symbolic representation of the model. For this reason, we are forced to present our model to NuSMV in the form of a program specification. The specification language NuSMV uses is called SMV [26].

We have chosen to specify our model in a way that allows us to take advantage of the model partitioning methods implemented in NuSMV. For example let us consider the following XML file:

```
<A>
        <B> </B>
        <B> </B>
</A>
```

Via our translation the skeleton of this XML document becomes the CTL model given in Figure 4.1. The specification of this model in the NuSMV input language is given in Figure 4.2. We will briefly explain the basic elements in this specification, in order to give the reader a sense of what is going on.

The model description that we feed to NuSMV is described using Boolean formulas. The set of states of the model is determined by the declaration of the state variable `s`. `s` is the scalar variable that takes a value from `0,1,2`. These values are the node identifiers in our model. With the help of the variable `labels` we describe the set of labels `A,B`. The variable `d` runs over the symbols `_a_,_up_,_down_,_left_,_right_` that represent the auxiliary labels $a, \uparrow, \downarrow, \leftarrow, \rightarrow$ in our CTL model. Each transition of the model is described by a pair consisting of a current state and a next state. Whether the pair is in the relation is again determined by a Boolean formula. For example the pair $((0, a)(0, d))$ is in the relation if `( s = 0 & d = _a_ -> (next(s) = 0 & next(d) != _a_))` holds. The transition relation of the model is expressed by a conjunction of such formulas, one for each pair. Finally, the labeling function of the model is described by the variable `l`, which specifies for each value of the variable `s` a set of values of `labels`.

While this seems to be a reasonable, and certainly linear encoding of the models, we were confronted with the problem of excessively big model descriptions. We believe that this can be avoided by directly preprocessing the model corresponding to the XML document into the internal NuSMV OBDD (Ordered Binary Decision Diagram) structures. OBDD structures have been argued to be on average logarithmically smaller then the models they describe [15].

```
MODULE main
VAR s : 0..2;
d : {_up_, _down_, _left_, _right_, _a_};
labels : {A,B};

TRANS
( s = 1 & d = _a_ -> (next(s) = 1 & next(d) != _a_) ) &
( s = 1 & d = _up_ -> ( (next(s) = 1 & next(d) = _a_) |
 (next(s) = 0 & next(d) = _up_) ) ) &
( s = 1 & d = _left_ -> (next(s) = 1 & next(d) = _a_) ) &
( s = 1 & d = _down_ -> (next(s) = 1 & next(d) = _a_) ) &
( s = 2 & d = _a_ -> (next(s) = 2 & next(d) != _a_) ) &
( s = 2 & d = _up_ -> ( (next(s) = 2 & next(d) = _a_) |
 (next(s) = 0 & next(d) = _up_) ) ) &
( s = 2 & d = _left_ ->( (next(s) = 2 & next(d) = _a_) |
 ( next(s) = 1 & next(d) = _left_) ) ) &
( s = 1 & d = _right_ -> ( (next(s) = 1 & next(d) = _a_) |
 (next(s) = 2 &  next(d) = _right_) ) )&
( s = 2 & d = _down_ -> (next(s) = 2 & next(d) = _a_) ) &
( s = 0 & d = _a_ -> (next(s) = 0 & next(d) != _a_) ) &
( s = 0 & d = _up_ -> (next(s) = 0 & next(d) = _a_) ) &
( s = 0 & d = _left_ -> (next(s) = 0 & next(d) = _a_) ) &
( s = 0 & d = _down_ -> ( (next(s) = 0 & next(d) = _a_) |
 (next(s) = 1 & next(d) = _down_) |
 (next(s) = 2 & next(d) = _down_) ) ) &
( s = 2 & d = _right_ -> (next(s) = 2 & next(d) = _a_) ) &
( s = 0 & d = _right_ -> (next(s) = 0 & next(d) = _a_) );

DEFINE
l := case
    s = 0 : A;
    s in {1,2} : B;
esac;
```

Figure 4.2: SMV specification of the CTL model given in Figure 4.1

4.1.2 Global model checking

NuSMV processing takes several computational steps, which can be activated through various commands. For instance, `read_model` instructs NuSMV to read the model (specified in the SMV specification language). After reading the model, `go` tells NuSMV to compile the model into its internal OBDD based representation, after which NuSMV is ready to check the formulas, using the command `check_spec`. For a detailed list of all NuSMV commands, see [11].

NuSMV was design to implement *local model checking*, i.e., given a formula, a model and a state of the model, check if the formula is satisfied at that particular state. We are interested in the *global model checking* problem, i.e., given a formula and a model, return the set of states that satisfy the formula. The latter is called the *truth set* of the formula. Internally, NuSMV has implemented global model checking. However, the interface does not provide the option for retrieving the truth set. Therefore, we had to extend the NuSMV interface with another command, `global_check_spec`, that gives us the truth set of CTL formulas. As will be clear to the reader, this command is a variant of the existing `check_spec` command. For reference, we give here the precise usage of the new command that we added.

```
usage: global_check_spec [-h] [-m | -o file] [-n number | -p "ctl-expr"]
   -h                   Prints the command usage.
   -m                   Pipes output through the program specified
                        by the "PAGER" environment variable if defined,
                        else through the UNIX command "more".
   -o file              Writes the generated output to "file".
   -n number            Checks only the SPEC with the given index number.
   -p "ctl-expr"        Checks only the given CTL formula and prints
                        out the set of states that satisfies the formula.
```

4.2 XCheck

XCheck is our prototype implementation of XPath query evaluation via CTL model checking. In implementing XCheck we aimed to achieve several goals:

- To check the practical feasibility of the translation provided in Chapter 3. We have a theoretical proof (cf. Theorem 3.1.10) that both translations, XML documents to CTL models and Core XPath queries to CTL formulas, are linear. However, whether they are suitable for practical use can only be tested experimentally.
- To evaluate to what extent Core XPath query evaluation gains from all the optimizations implemented in NuSMV.
- To compare the performance of model checking with that of two existing linear time query evaluation systems, namely MacMill [8] and XMLTaskForce XPath [20].

XCheck allowed us to perform the tests we were interested in.

4.2.1 Technical specifications

Implementation

XCheck is written in Perl and consists mainly of two translation subroutines, one for translating the XML documents into NuSMV input format, and another for translating Core XPath queries to CTL formulas. Besides implementing these translations, XCheck contains a subroutine that runs NuSMV and a subroutine that interprets the truth set of CTL formula as the result of XML query evaluation.

sub translate_model This subroutine implements the model translation. It takes as input an XML document and generates an CTL model conform translation μ_1 and μ_2 (Definition 3.1.1 and 3.1.6). The resulting model is described in the SMV language (cf. Section 4.1.1). The subroutine uses the perl module XML::Twig, designed and written be Michel Rodriguez [29]. We execute a depth first traversal of the XML tree to translate the relation between the nodes.

sub translate_formula This subroutine executes the Core XPath query translation into CTL formula conform translations τ_1, ω_1, τ_2 and ω_2 (Definitions 3.1.3 and 3.1.7, respectively). Our implementation of the query translation rewrites a Core XPath query directly to a CTL formula, thus avoiding the intermediate representation in many dimensional CTL. Besides this, the subroutine follows precisely the theoretical translation. The subroutine calls two additional ones, **sub tau** and **sub mu**, which have the subtasks of translating the location paths and predicates, respectively.

sub nusmv This subroutine calls NuSMV. It runs the model checker on the generated model and formula, and returns the truth set of the formula, i.e., the set of states satisfying the formula.

XCheck keeps track of the CPU time elapsed while running each translation and each NuSMV command. The source code of above mentioned subroutines is included in Appendix 1.

As it stands, XCheck is not suitable for interactive, real-time XPath query evaluation.

Usage

```
system_promt> ./XCheck [-tc] [-d doc.xml] [-q "query"]
```

OPTIONS:

`-d doc.xml`
: extracts the skeleton of the XML document `doc.xml`, and translates it into `doc.smv`. `doc.xml` is any XML document, conform the W3C specifications [16]. `doc.smv` is the NuSMV specification of the CTL model obtained by using the translations μ_1 and μ_2 (Definitions 3.1.1,3.1.6).

`-q "query"`
: translates `query` into CTL formula conform translation τ_1 and τ_2 (Definitions 3.1.3,3.1.7).
`query` is a Core XPath query conform Definition 2.2.1.

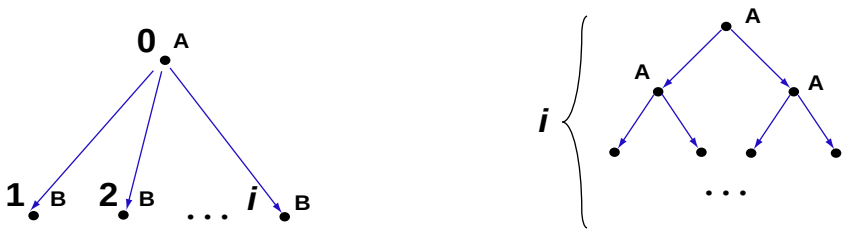
`-t`
: reports the elapsed CPU time

`-c`
: must be used in combination with `-d doc.xml -q "query"`. Invokes NuSMV to build an internal representation of (the translation of) `doc.xml`, and the to perform global model checking on the CTL translation of `"query"`.

4.3 Experimental results

We performed four experiments.

Experiment 1 We tested the time of translating the XML document into NuSMV input format. By Corollary 3.1.11, we know that the translation time should be linear in the size of the document. The goal of the experiment is two-fold: firstly, to confirm that the implementation of the translation is indeed linear, and secondly, to see how efficient it is.



A root with i children

A complete tree of depth i

Figure 4.3: The tests models

Experiment 2 We tested the time it takes NuSMV to read the model, and to convert it into its internal, OBDD based represenation. In practice, this preprocessing of the model (which also includes the translation mentioned above) can be done beforehand, rather than real-time. This means that the performance is of less importance than for instance that of the query translation. However, the efficiency of preprosessing is still of interest, and the experiment is meant to measure it. At the same time, the experiment also might give us more information about the behavior of NuSMV, when given as input XML documents according to our representation.

Experiment 3 We tested the time of translating Core XPath queries into CTL formulas. Again, by Corollary 3.1.11, we know that the translation time should be linear in the size of the query. Then, the goals of the experiment are: to confirm that the implementation of the translation is indeed linear and to see how efficient it is.

Experiment 4 We tested the time it takes NuSMV to perform the actual model checking. This does not include the time of translating the document and the query, nor the time it takes NuSMV to compile the model. We tested the two other query evaluation systems (MacMill and XMLTaskForce XPath) with the same input, in order to compare the running times.

Experimental set-up

For our experiments we use artificially generated data. Our motivation is the following: first, real XML documents contain a lot of information, while we are interested only in the navigational part of the document. The skeleton of a typical XML document is about 10% of the size of the document. It is therefore hard to obtain enough real data to run the tests with preferred granularity. This is also because there are no benchmarks for testing Core XPath queries (as opposed to full XPath). By using generated data, we can also systematically test the influence of particular parameters on the performance (e.g., the *branching factor* of the XML document).

The queries and document we use are similar to the ones that were used to evaluate XMLTaskForce XPath. In [20], the authors motivate the use of this type of generated data.

Let $i \geq 0$. We are going to use two different parametric documents:

1. Let $doc_1(i) = \langle a \rangle \langle b/ \rangle^i \langle /a \rangle$, where, $\langle b/ \rangle^0 = \epsilon$ (the empty document) and, for $i > 0$, $\langle b/ \rangle^i = \langle b/ \rangle \langle b/ \rangle^{i-1}$. Recall that $\langle x/ \rangle$ is a shorthand for $\langle x \rangle \langle /x \rangle$. Hence, $doc_1(i)$ is a tree with $i+1$ nodes: a root labeled with a and i children labeled with b. See Figure 4.3.

2. Let $doc_2(i)$ be such that $doc_2(0) = \langle a/ \rangle$ and, for $i > 0$, $doc_2(i) = \langle a \rangle doc_2(i-1) doc_2(i-1) \langle /a \rangle$. Hence, $doc_2(i)$ is a complete binary tree with $2^{i+1} - 1$ nodes all labeled with a. See Figure 4.3.

We will experiment with four different parametric queries, each focusing on different pairs of opposite axes:

Table 4.1: Document translation and model compilation time (n=18)

$doc_1(i)$	translation	reading	building
500	0.1403 ± 0.0059	0.0861 ± 0.0466	3.1039 ± 0.3108
1000	0.2850 ± 0.0166	0.1822 ± 0.0212	13.5850 ± 1.0681
1500	0.4276 ± 0.0233	0.2683 ± 0.0285	32.3317 ± 1.1510
2000	0.5764 ± 0.0197	0.3478 ± 0.0279	56.6317 ± 1.2893
2500	0.7149 ± 0.0303	0.4422 ± 0.0243	94.4044 ± 2.0908
3000	0.8560 ± 0.0474	0.5156 ± 0.0240	126.3800 ± 2.4331
3500	1.0427 ± 0.1003	0.5167 ± 0.0536	178.1006 ± 4.7429
4000	1.1727 ± 0.0584	0.5989 ± 0.0943	230.6994 ± 3.5183
4500	1.3014 ± 0.1315	0.6617 ± 0.0505	293.7317 ± 5.4735
5000	1.4168 ± 0.0739	0.7300 ± 0.0428	354.5283 ± 4.4143

$doc_2(i)$	translation	reading	building
5	0.0203 ± 0.0009	0.0239 ± 0.0122	0.1022 ± 0.0201
6	0.0383 ± 0.0040	0.0300 ± 0.0206	0.2856 ± 0.1286
7	0.0751 ± 0.0047	0.0494 ± 0.0175	0.9561 ± 0.0276
8	0.1506 ± 0.0054	0.0911 ± 0.0166	3.3428 ± 0.1680
9	0.3069 ± 0.0119	0.1739 ± 0.0156	13.9000 ± 0.8593
10	0.6183 ± 0.0249	0.3356 ± 0.0259	51.7911 ± 2.2384

1. Let $q_1(i) =$ `descendant_or_self::A/`$\underbrace{path/\ \ldots\ /path}_{i \text{ times}}$, where

   $path$=`child::B/parent::A` and $i \geq 0$. We will call the number of path steps in a query the *length* of the query. The length of the query $q_1(i)$ is $2i + 1$.

2. Let $q_2(i) =$ `/descendant::`$A/\underbrace{path/\ \ldots\ /path}_{i \text{ times}}$, where

   $path$=`following_sibling::B/preceding\_sibling::B` and $i \geq 0$. The length of the query $q_2(i)$ is $2i + 1$.

3. Let $q_3(i) =$ `/descendant_or_self::A/`$\underbrace{path/\ \ldots\ /path}_{i \text{ times}}$, where

   $path$=`descendant::B/ancestor::A` and $i \geq 0$. Hence, $q_3(i)$ is a query of length $2i + 1$.

4. Let $q_4(i) =$ `/descendant::A/`$\underbrace{path/\ \ldots\ /path}_{i \text{ times}}$, where

   $path$=`/following::A/preceding::A` and $i \geq 0$. The query $q_4(i)$ has length $2i + 1$.

The experiments are run on a 1.50GHz Intel(R) Pentium(R) 4 machine, with 1,5GB RAM, running Linux version 2.4.21-ict1. All reported times are in seconds CPU time.

Results

A brief description of each test case follows including the results. In each case, the times are given in seconds. In the result tables we also included the standard deviation for each of the results, to indicate the reliability of the results, and n indicates the number of experiments ran.

Experiment 1 and 2 Testing the document translation and model compilation time.

Table 4.1 depicts the translation time of the documents generated by $doc_1(i)$ and $doc_2(j)$, as well as the time it took NuSMV to read the model and compile it into its internal, OBDD

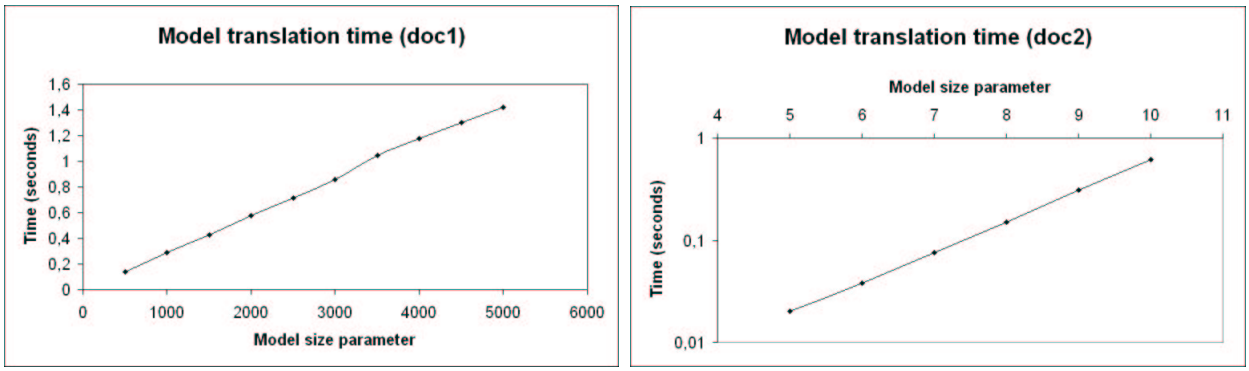


Figure 4.4: Document translation time

Table 4.2: Query translation times (n=10)

i	$q_1(i)$	$q_2(i)$	$q_3(i)$	$q_4(i)$
100	0.0008 ± 0.0002	0.0010 ± 0.0002	0.0008 ± 0.0001	0.0012 ± 0.0004
200	0.0020 ± 0.0004	0.0029 ± 0.0009	0.0023 ± 0.0003	0.0052 ± 0.0019
300	0.0041 ± 0.0009	0.0065 ± 0.0014	0.0049 ± 0.0011	0.0131 ± 0.0068
400	0.0077 ± 0.0013	0.0120 ± 0.0026	0.0091 ± 0.0022	0.0272 ± 0.0126
500	0.0122 ± 0.0034	0.0196 ± 0.0046	0.0152 ± 0.0044	0.0492 ± 0.0162
600	0.0169 ± 0.0065	0.0317 ± 0.0089	0.0228 ± 0.0090	0.0789 ± 0.0157
700	0.0267 ± 0.0083	0.0457 ± 0.0165	0.0349 ± 0.0129	0.1107 ± 0.0183
800	0.0348 ± 0.0140	0.0617 ± 0.0187	0.0478 ± 0.0178	0.1675 ± 0.0401
900	0.0461 ± 0.0175	0.0805 ± 0.0274	0.0625 ± 0.0195	0.2166 ± 0.0243
1000	0.0586 ± 0.0222	0.1073 ± 0.0362	0.0813 ± 0.0266	0.2465 ± 0.0376

based representation. On the Y-axis is indicated the parameter which determines the size of the generated models.

The translation times are as expected: in both cases, the translation time grows linearly with respect to the size of the document (cf. Figure 4.4). Note that the size of $doc_2(i)$ is exponential in i.

The time taken by NuSMV to build the internal OBDD-based representation of the model is high. As we mentioned before, this can be considered part of preprocessing the data. However, it might suggest that the OBDD-based representation is large. So far, we have not investigated the size of the OBDD-based representations, but we consider this necessary for further work on our approach.

During our experiments, we discovered that NuSMV has a limited processing capacity for large input files. It cannot cope with very large program specifications. In the case of our experiments, NuSMV could not read and process $\mathrm{doc}_1(i)$ approximitely of size $i \geq 60000$, or $\mathrm{doc}_2(i)$ approximitely of size $i \geq 14$.

Experiment 3 Testing the query translation time.

The results of our query translation experiment are reported in Table 4.2. On the X-axis, the different types of queries are given. On the Y-axis, the parameter in given that determines the size of the queries. Note that the size of the queries is linear with respect to the parameter. We would therefore expect that the values in the matrix are linear with respect to this parameter as well.

As can be seen from Figure 4.5, the translation time is quadratic in the size of the input. This indicates that there is a problem with the implementation of the translation. The

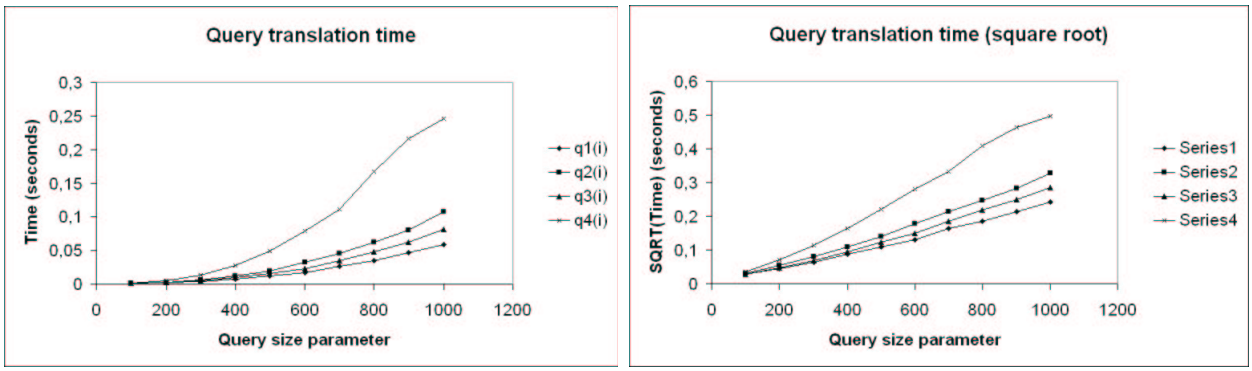


Figure 4.5: Query translation time

cause might lie in the way Perl implements regular expression matching. The worst-case complexity of regular expression matching in Perl is exponential in the size of the input, due to its enriched syntax for regular expressions [18].

Experiment 4 Testing the query evaluation time.

In Table 4.3, 4.4 and 4.5, every cell contains three lines. The first indicates the time NuSMV needed to check the translated query on the compiled, translated document. The second and third entries indicate the time needed by MacMill and XMLTaskForce XPath, respectively, to evaluate the query.

For optimal precision, we performed the model checking 100 times, divided in 10 jobs. Of every job, the total time was measured, and divided by 10. The average of the resulting 10 numbers is reported in the tables, as well as the standard deviation computed on these 10 numbers.

In the case of MacMill and XMLTaskForce XPath, each experiment was simply performed 10 times.

The model checking time is slightly higher than the time needed by MacMill and XMLTaskForce XPath to perform the query evaluation. Perhaps this could be explained by the fact that, while our translation is linear, the size of the translated model is high.

Figure 4.6 contains several graphs that represent the observed model checking times. As one can clearly see, NuSMV model checking is linear in the size of the query, and almost linear in the document size.

Incidentally, note the remarkable shape of the curve for model checking q_2 queries on doc_1 document. Perhaps, some features of NuSMV play a role in this, but we could not tell.

Interpretation of the results and suggestions for improvements

- We noticed that there is a problem with our implementation of query translation. Our implementation runs in quadratic time in the size of the query, although our translation admits a linear implementation. We suspect that this problem is caused by Perl's implementation of regular expression matching that we use in our implementation. We believe that a linear implementation of the translation is easy to achieve by being more careful in this respect.
- What needs more attention is the setup of our experiments. It is always important to evaluate theory in practice. So far all the experiments we performed involved artificially generated XML documents and queries. Testing on real data is essential to get a clear picture of the performance of XCheck. This remains future work.
- We consider it absolutely necessary to perform more experiments that explicitly measure the size of the internal OBDD-based representation versus the input XML document.

Table 4.3: Model checking time for $q_1(i)$ queries on $doc_1(j)$ documents

$doc_1(i)\backslash q_1(j)$	5	10	15	20	25	30	35	40	45
500	0.0025 ± 0.0014	0.0039 ± 0.0011	0.0057 ± 0.0016	0.0067 ± 0.0021	0.0080 ± 0.0023	0.0097 ± 0.0025	0.0116 ± 0.0023	0.0130 ± 0.0027	0.0158 ± 0.0023
	0.0010 ± 0.0001	0.0010 ± 0.0001	0.0010 ± 0.0001	0.0010 ± 0.0001	0.0010 ± 0.0001	0.0011 ± 0.0001	0.0011 ± 0.0001	0.0011 ± 0.0001	0.0011 ± 0.0001
	0.0125 ± 0.0006	0.0131 ± 0.0002	0.0137 ± 0.0001	0.0147 ± 0.0006	0.0152 ± 0.0001	0.0182 ± 0.0091	0.0169 ± 0.0001	0.0179 ± 0.0016	0.0184 ± 0.0006
1000	0.0028 ± 0.0021	0.0044 ± 0.0017	0.0061 ± 0.0026	0.0081 ± 0.0022	0.0097 ± 0.0016	0.0115 ± 0.0030	0.0132 ± 0.0013	0.0146 ± 0.0025	0.0166 ± 0.0023
	0.0009 ± 0.0001	0.0009 ± 0.0001	0.0009 ± 0.0001	0.0009 ± 0.0000	0.0010 ± 0.0000	0.0010 ± 0.0000	0.0010 ± 0.0001	0.0010 ± 0.0001	0.0010 ± 0.0000
	0.0145 ± 0.0006	0.0158 ± 0.0002	0.0170 ± 0.0006	0.0185 ± 0.0007	0.0198 ± 0.0008	0.0213 ± 0.0007	0.0225 ± 0.0002	0.0238 ± 0.0002	0.0255 ± 0.0028
1500	0.0034 ± 0.0034	0.0043 ± 0.0039	0.0070 ± 0.0041	0.0089 ± 0.0038	0.0104 ± 0.0040	0.0127 ± 0.0028	0.0149 ± 0.0036	0.0166 ± 0.0030	0.0192 ± 0.0023
	0.0009 ± 0.0001	0.0009 ± 0.0000	0.0009 ± 0.0001	0.0010 ± 0.0000	0.0010 ± 0.0000	0.0010 ± 0.0000	0.0010 ± 0.0000	0.0010 ± 0.0000	0.0013 ± 0.0016
	0.0163 ± 0.0001	0.0184 ± 0.0002	0.0202 ± 0.0007	0.0224 ± 0.0008	0.0241 ± 0.0007	0.0265 ± 0.0006	0.0284 ± 0.0008	0.0305 ± 0.0010	0.0325 ± 0.0011
2000	0.0038 ± 0.0052	0.0057 ± 0.0060	0.0077 ± 0.0061	0.0098 ± 0.0054	0.0116 ± 0.0061	0.0141 ± 0.0045	0.0161 ± 0.0053	0.0180 ± 0.0059	0.0211 ± 0.0056
	0.0009 ± 0.0000	0.0009 ± 0.0000	0.0010 ± 0.0001	0.0012 ± 0.0012	0.0010 ± 0.0000	0.0010 ± 0.0000	0.0011 ± 0.0000	0.0011 ± 0.0000	0.0011 ± 0.0001
	0.0182 ± 0.0003	0.0210 ± 0.0009	0.0250 ± 0.0125	0.0258 ± 0.0002	0.0283 ± 0.0004	0.0315 ± 0.0007	0.0363 ± 0.0127	0.0364 ± 0.0007	0.0389 ± 0.0010
2500	0.0044 ± 0.0083	0.0066 ± 0.0074	0.0089 ± 0.0064	0.0113 ± 0.0076	0.0137 ± 0.0073	0.0164 ± 0.0077	0.0174 ± 0.0078	0.0217 ± 0.0087	0.0229 ± 0.0079
	0.0009 ± 0.0000	0.0010 ± 0.0002	0.0010 ± 0.0000	0.0010 ± 0.0000	0.0010 ± 0.0000	0.0010 ± 0.0000	0.0011 ± 0.0001	0.0011 ± 0.0000	0.0011 ± 0.0000
	0.0205 ± 0.0001	0.0249 ± 0.0036	0.0270 ± 0.0009	0.0307 ± 0.0009	0.0342 ± 0.0007	0.0384 ± 0.0007	0.0418 ± 0.0013	0.0489 ± 0.0034	0.0482 ± 0.0010
3000	0.0056 ± 0.0116	0.0079 ± 0.0107	0.0107 ± 0.0108	0.0130 ± 0.0114	0.0160 ± 0.0113	0.0182 ± 0.0108	0.0210 ± 0.0101	0.0227 ± 0.0110	0.0315 ± 0.0103
	0.0010 ± 0.0000	0.0010 ± 0.0000	0.0010 ± 0.0001	0.0010 ± 0.0000	0.0010 ± 0.0000	0.0011 ± 0.0001	0.0011 ± 0.0000	0.0011 ± 0.0000	0.0011 ± 0.0000
	0.0223 ± 0.0002	0.0267 ± 0.0008	0.0300 ± 0.0007	0.0349 ± 0.0044	0.0378 ± 0.0007	0.0429 ± 0.0007	0.0470 ± 0.0008	0.0554 ± 0.0247	0.0548 ± 0.0012
3500	0.0087 ± 0.0290	0.0120 ± 0.0290	0.0148 ± 0.0277	0.0174 ± 0.0295	0.0208 ± 0.0292	0.0230 ± 0.0305	0.0264 ± 0.0288	0.0280 ± 0.0277	0.0325 ± 0.0279
	0.0010 ± 0.0000	0.0010 ± 0.0000	0.0010 ± 0.0000	0.0010 ± 0.0000	0.0011 ± 0.0000	0.0011 ± 0.0001	0.0011 ± 0.0000	0.0011 ± 0.0000	0.0011 ± 0.0000
	0.0249 ± 0.0003	0.0290 ± 0.0003	0.0337 ± 0.0002	0.0377 ± 0.0008	0.0426 ± 0.0009	0.0474 ± 0.0008	0.0531 ± 0.0010	0.0578 ± 0.0011	0.0627 ± 0.0054
4000	0.0099 ± 0.0360	0.0130 ± 0.0359	0.0153 ± 0.0371	0.0181 ± 0.0372	0.0215 ± 0.0369	0.0240 ± 0.0359	0.0266 ± 0.0368	0.0292 ± 0.0351	0.0322 ± 0.0359
	0.0014 ± 0.0018	0.0015 ± 0.0019	0.0014 ± 0.0020	0.0011 ± 0.0000	0.0013 ± 0.0012	0.0011 ± 0.0000	0.0011 ± 0.0000	0.0011 ± 0.0000	0.0011 ± 0.0000
	0.0268 ± 0.0006	0.0316 ± 0.0006	0.0369 ± 0.0007	0.0414 ± 0.0009	0.0467 ± 0.0010	0.0526 ± 0.0010	0.0589 ± 0.0015	0.0643 ± 0.0011	0.0685 ± 0.0008
4500	0.0111 ± 0.0400	0.0144 ± 0.0426	0.0170 ± 0.0387	0.0202 ± 0.0378	0.0232 ± 0.0385	0.0268 ± 0.0381	0.0297 ± 0.0396	0.0334 ± 0.0405	0.0363 ± 0.0357
	0.0010 ± 0.0000	0.0010 ± 0.0000	0.0010 ± 0.0000	0.0011 ± 0.0001	0.0011 ± 0.0000	0.0011 ± 0.0000	0.0011 ± 0.0000	0.0011 ± 0.0000	0.0012 ± 0.0000
	0.0293 ± 0.0001	0.0348 ± 0.0003	0.0411 ± 0.0010	0.0466 ± 0.0036	0.0526 ± 0.0012	0.0586 ± 0.0010	0.0666 ± 0.0030	0.0726 ± 0.0008	0.0774 ± 0.0009
5000	0.0117 ± 0.0438	0.0152 ± 0.0442	0.0187 ± 0.0445	0.0239 ± 0.0549	0.0249 ± 0.0458	0.0285 ± 0.0433	0.0333 ± 0.0462	0.0372 ± 0.0477	0.0393 ± 0.0428
	0.0010 ± 0.0000	0.0010 ± 0.0001	0.0011 ± 0.0000	0.0011 ± 0.0000	0.0012 ± 0.0006	0.0011 ± 0.0000	0.0011 ± 0.0000	0.0012 ± 0.0001	0.0012 ± 0.0000
	0.0312 ± 0.0005	0.0376 ± 0.0010	0.0442 ± 0.0009	0.0495 ± 0.0006	0.0568 ± 0.0016	0.0637 ± 0.0010	0.0719 ± 0.0014	0.0788 ± 0.0012	0.0843 ± 0.0012

Table 4.4: Model checking time for $q_2(i)$ queries on $doc_1(j)$ documents

$doc_1(i)\backslash q_2(j)$	5	10	15	20	25	30	35	40	45
500	0.0968 ± 0.0117	0.1009 ± 0.0143	0.1001 ± 0.0139	0.1047 ± 0.0108	0.1122 ± 0.0136	0.1147 ± 0.0121	0.1169 ± 0.0152	0.1229 ± 0.0101	0.1258 ± 0.0153
	0.0014 ± 0.0001	0.0014 ± 0.0001	0.0014 ± 0.0001	0.0015 ± 0.0001	0.0015 ± 0.0001	0.0016 ± 0.0001	0.0016 ± 0.0001	0.0017 ± 0.0001	0.0017 ± 0.0001
	0.0129 ± 0.0001	0.0142 ± 0.0006	0.0153 ± 0.0002	0.0165 ± 0.0001	0.0179 ± 0.0008	0.0190 ± 0.0006	0.0200 ± 0.0008	0.0211 ± 0.0002	0.0221 ± 0.0002
1000	0.1952 ± 0.0135	0.2064 ± 0.0117	0.2150 ± 0.0167	0.2175 ± 0.0148	0.2225 ± 0.0152	0.2284 ± 0.0200	0.2346 ± 0.0208	0.2324 ± 0.0196	0.2194 ± 0.0678
	0.0018 ± 0.0001	0.0018 ± 0.0000	0.0019 ± 0.0001	0.0019 ± 0.0001	0.0019 ± 0.0001	0.0021 ± 0.0001	0.0020 ± 0.0001	0.0022 ± 0.0000	0.0021 ± 0.0000
	0.0155 ± 0.0003	0.0174 ± 0.0001	0.0209 ± 0.0050	0.0220 ± 0.0002	0.0245 ± 0.0008	0.0263 ± 0.0008	0.0283 ± 0.0007	0.0307 ± 0.0006	0.0325 ± 0.0010
1500	0.4723 ± 0.0461	0.4885 ± 0.0416	0.4816 ± 0.0392	0.5021 ± 0.0402	0.5172 ± 0.0342	0.5142 ± 0.0473	0.5242 ± 0.0351	0.5283 ± 0.0445	0.5462 ± 0.0556
	0.0025 ± 0.0008	0.0025 ± 0.0001	0.0025 ± 0.0001	0.0025 ± 0.0001	0.0026 ± 0.0001	0.0028 ± 0.0001	0.0027 ± 0.0002	0.0030 ± 0.0013	0.0028 ± 0.0001
	0.0182 ± 0.0007	0.0210 ± 0.0009	0.0246 ± 0.0003	0.0276 ± 0.0008	0.0309 ± 0.0009	0.0338 ± 0.0008	0.0367 ± 0.0009	0.0403 ± 0.0007	0.0429 ± 0.0008
2000	1.0485 ± 0.0198	1.0367 ± 0.1094	1.0700 ± 0.0196	1.0542 ± 0.1221	1.0920 ± 0.0507	1.0920 ± 0.1150	1.1069 ± 0.0400	1.1432 ± 0.0648	1.1240 ± 0.0503
	0.0029 ± 0.0001	0.0030 ± 0.0001	0.0030 ± 0.0001	0.0034 ± 0.0018	0.0032 ± 0.0003	0.0034 ± 0.0001	0.0034 ± 0.0012	0.0038 ± 0.0013	0.0033 ± 0.0001
	0.0212 ± 0.0037	0.0243 ± 0.0002	0.0291 ± 0.0002	0.0330 ± 0.0005	0.0374 ± 0.0005	0.0418 ± 0.0043	0.0452 ± 0.0009	0.0501 ± 0.0009	0.0534 ± 0.0011
2500	2.7835 ± 0.0660	2.7570 ± 0.1348	2.8271 ± 0.1446	2.8354 ± 0.1672	2.8596 ± 0.2039	2.9016 ± 0.0864	2.9122 ± 0.0882	2.8389 ± 0.1351	2.9355 ± 0.1565
	0.0038 ± 0.0012	0.0036 ± 0.0001	0.0037 ± 0.0001	0.0038 ± 0.0006	0.0037 ± 0.0001	0.0042 ± 0.0007	0.0038 ± 0.0001	0.0041 ± 0.0001	0.0040 ± 0.0001
	0.0234 ± 0.0006	0.0285 ± 0.0010	0.0351 ± 0.0016	0.0394 ± 0.0009	0.0454 ± 0.0008	0.0513 ± 0.0044	0.0561 ± 0.0015	0.0683 ± 0.0024	0.0666 ± 0.0013
3000	2.4553 ± 0.1205	2.5365 ± 0.1615	2.6655 ± 0.1437	2.6196 ± 0.0707	2.6129 ± 0.0632	2.6403 ± 0.1450	2.6712 ± 0.0987	2.6622 ± 0.0685	2.7117 ± 0.1752
	0.0041 ± 0.0010	0.0041 ± 0.0001	0.0041 ± 0.0003	0.0042 ± 0.0001	0.0043 ± 0.0001	0.0047 ± 0.0002	0.0044 ± 0.0001	0.0047 ± 0.0002	0.0045 ± 0.0001
	0.0260 ± 0.0008	0.0321 ± 0.0009	0.0395 ± 0.0009	0.0455 ± 0.0006	0.0526 ± 0.0015	0.0587 ± 0.0015	0.0651 ± 0.0028	0.0725 ± 0.0021	0.0769 ± 0.0018
3500	3.2981 ± 0.1670	3.3360 ± 0.1666	3.3571 ± 0.1698	3.3413 ± 0.1677	3.3557 ± 0.1200	3.4102 ± 0.1633	3.4409 ± 0.2380	3.3729 ± 0.1647	3.5014 ± 0.1220
	0.0047 ± 0.0001	0.0047 ± 0.0001	0.0048 ± 0.0001	0.0048 ± 0.0001	0.0048 ± 0.0001	0.0057 ± 0.0007	0.0050 ± 0.0001	0.0051 ± 0.0001	0.0078 ± 0.0141
	0.0294 ± 0.0036	0.0360 ± 0.0007	0.0431 ± 0.0015	0.0509 ± 0.0010	0.0592 ± 0.0034	0.0655 ± 0.0011	0.0723 ± 0.0011	0.0824 ± 0.0042	0.0880 ± 0.0014
4000	3.5867 ± 0.1859	3.6363 ± 0.1408	3.5753 ± 0.0567	3.5175 ± 0.0905	3.5677 ± 0.0899	3.6143 ± 0.1460	3.5971 ± 0.1884	3.6208 ± 0.1990	3.7258 ± 0.2222
	0.0059 ± 0.0028	0.0064 ± 0.0033	0.0072 ± 0.0040	0.0055 ± 0.0012	0.0058 ± 0.0019	0.0063 ± 0.0002	0.0055 ± 0.0001	0.0056 ± 0.0001	0.0062 ± 0.0006
	0.0315 ± 0.0011	0.0394 ± 0.0008	0.0473 ± 0.0008	0.0568 ± 0.0012	0.0654 ± 0.0013	0.0742 ± 0.0031	0.0810 ± 0.0014	0.0918 ± 0.0016	0.0984 ± 0.0019
4500	7.6832 ± 0.2207	7.6246 ± 0.2760	7.7794 ± 0.2159	7.6734 ± 0.1963	7.6409 ± 0.2151	7.9226 ± 0.3278	7.8158 ± 0.4348	7.9990 ± 0.3123	7.8269 ± 0.3446
	0.0058 ± 0.0006	0.0057 ± 0.0002	0.0058 ± 0.0005	0.0058 ± 0.0002	0.0058 ± 0.0002	0.0068 ± 0.0003	0.0060 ± 0.0002	0.0061 ± 0.0002	0.0067 ± 0.0002
	0.0346 ± 0.0010	0.0438 ± 0.0011	0.0527 ± 0.0009	0.0636 ± 0.0011	0.0735 ± 0.0008	0.0826 ± 0.0013	0.0917 ± 0.0019	0.1044 ± 0.0021	0.1113 ± 0.0017
5000	8.2702 ± 0.2774	8.3485 ± 0.2064	8.2614 ± 0.3283	8.2711 ± 0.3099	8.3545 ± 0.2226	8.7254 ± 0.4070	8.4415 ± 0.4208	8.4225 ± 0.4608	8.6688 ± 0.5031
	0.0062 ± 0.0002	0.0062 ± 0.0002	0.0063 ± 0.0004	0.0064 ± 0.0006	0.0063 ± 0.0004	0.0076 ± 0.0004	0.0066 ± 0.0002	0.0067 ± 0.0003	0.0074 ± 0.0002
	0.0367 ± 0.0004	0.0470 ± 0.0010	0.0648 ± 0.0469	0.0687 ± 0.0009	0.0798 ± 0.0012	0.0905 ± 0.0038	0.0998 ± 0.0020	0.1143 ± 0.0042	0.1223 ± 0.0023

Table 4.5: Model checking time for $q_3(i)$ and $q_4(i)$ queries on $doc_2(j)$ documents

$doc_2(i)\backslash q_3(j)$	5	10	15	20	25	30	35	40	45
5	0.0075 ± 0.0019	0.0099 ± 0.0011	0.0126 ± 0.0021	0.0148 ± 0.0013	0.0180 ± 0.0016	0.0210 ± 0.0025	0.0214 ± 0.0037	0.0244 ± 0.0034	0.0272 ± 0.0043
	0.0010 ± 0.0001	0.0011 ± 0.0001	0.0011 ± 0.0001	0.0011 ± 0.0001	0.0012 ± 0.0001	0.0013 ± 0.0001	0.0013 ± 0.0001	0.0013 ± 0.0001	0.0014 ± 0.0001
	0.0110 ± 0.0007	0.0111 ± 0.0002	0.0115 ± 0.0003	0.0117 ± 0.0001	0.0123 ± 0.0009	0.0125 ± 0.0009	0.0127 ± 0.0001	0.0133 ± 0.0010	0.0134 ± 0.0002
6	0.0128 ± 0.0021	0.0159 ± 0.0020	0.0185 ± 0.0019	0.0218 ± 0.0023	0.0254 ± 0.0021	0.0280 ± 0.0031	0.0312 ± 0.0030	0.0343 ± 0.0033	0.0371 ± 0.0053
	0.0010 ± 0.0001	0.0012 ± 0.0006	0.0011 ± 0.0001	0.0012 ± 0.0001	0.0012 ± 0.0001	0.0013 ± 0.0000	0.0013 ± 0.0000	0.0014 ± 0.0002	0.0014 ± 0.0001
	0.0113 ± 0.0001	0.0119 ± 0.0009	0.0121 ± 0.0001	0.0125 ± 0.0001	0.0132 ± 0.0008	0.0135 ± 0.0001	0.0139 ± 0.0001	0.0144 ± 0.0002	0.0147 ± 0.0001
7	0.0247 ± 0.0013	0.0283 ± 0.0027	0.0333 ± 0.0027	0.0353 ± 0.0030	0.0393 ± 0.0028	0.0445 ± 0.0027	0.0467 ± 0.0035	0.0500 ± 0.0067	0.0541 ± 0.0060
	0.0011 ± 0.0001	0.0012 ± 0.0001	0.0012 ± 0.0000	0.0012 ± 0.0001	0.0013 ± 0.0001	0.0014 ± 0.0001	0.0014 ± 0.0000	0.0016 ± 0.0007	0.0015 ± 0.0000
	0.0119 ± 0.0006	0.0125 ± 0.0002	0.0133 ± 0.0006	0.0146 ± 0.0035	0.0146 ± 0.0001	0.0156 ± 0.0007	0.0161 ± 0.0002	0.0171 ± 0.0007	0.0175 ± 0.0002
8	0.0489 ± 0.0076	0.0551 ± 0.0058	0.0667 ± 0.0143	0.0636 ± 0.0055	0.0684 ± 0.0058	0.0785 ± 0.0072	0.0827 ± 0.0081	0.0802 ± 0.0066	0.0805 ± 0.0564
	0.0012 ± 0.0000	0.0013 ± 0.0001	0.0013 ± 0.0001	0.0014 ± 0.0001	0.0014 ± 0.0000	0.0015 ± 0.0000	0.0017 ± 0.0006	0.0016 ± 0.0001	0.0017 ± 0.0001
	0.0134 ± 0.0007	0.0145 ± 0.0007	0.0156 ± 0.0002	0.0168 ± 0.0002	0.0183 ± 0.0007	0.0197 ± 0.0007	0.0214 ± 0.0010	0.0223 ± 0.0007	0.0237 ± 0.0008
9	0.1191 ± 0.0307	0.1248 ± 0.0282	0.1342 ± 0.0280	0.1360 ± 0.0284	0.1430 ± 0.0282	0.1474 ± 0.0158	0.1539 ± 0.0252	0.1599 ± 0.0259	0.1695 ± 0.0247
	0.0014 ± 0.0000	0.0014 ± 0.0001	0.0015 ± 0.0000	0.0016 ± 0.0000	0.0017 ± 0.0001	0.0021 ± 0.0016	0.0019 ± 0.0001	0.0019 ± 0.0000	0.0019 ± 0.0001
	0.0160 ± 0.0002	0.0183 ± 0.0002	0.0206 ± 0.0006	0.0227 ± 0.0004	0.0260 ± 0.0011	0.0300 ± 0.0007	0.0317 ± 0.0010	0.0343 ± 0.0009	0.0370 ± 0.0012
10	0.7312 ± 0.1016	0.7657 ± 0.0360	0.7827 ± 0.0361	0.7771 ± 0.0274	0.7915 ± 0.0259	0.7975 ± 0.0309	0.7821 ± 0.1070	0.8208 ± 0.0217	0.8511 ± 0.0505
	0.0021 ± 0.0000	0.0022 ± 0.0000	0.0025 ± 0.0001	0.0026 ± 0.0001	0.0025 ± 0.0001	0.0026 ± 0.0000	0.0027 ± 0.0001	0.0027 ± 0.0001	0.0028 ± 0.0001
	0.0224 ± 0.0013	0.0409 ± 0.0887	0.0314 ± 0.0011	0.0364 ± 0.0006	0.0421 ± 0.0009	0.0476 ± 0.0011	0.0519 ± 0.0006	0.0565 ± 0.0011	0.0601 ± 0.0025

$doc_2(i)\backslash q_4(j)$	5	10	15	20	25	30	35	40	45
5	0.0154 ± 0.0021	0.0220 ± 0.0021	0.0286 ± 0.0019	0.0352 ± 0.0031	0.0432 ± 0.0040	0.0479 ± 0.0061	0.0540 ± 0.0112	0.0628 ± 0.0101	0.0653 ± 0.0091
	0.0012 ± 0.0001	0.0014 ± 0.0002	0.0016 ± 0.0001	0.0018 ± 0.0007	0.0019 ± 0.0001	0.0020 ± 0.0000	0.0061 ± 0.0245	0.0024 ± 0.0001	0.0027 ± 0.0002
	0.0111 ± 0.0001	0.0117 ± 0.0007	0.0122 ± 0.0008	0.0127 ± 0.0001	0.0131 ± 0.0002	0.0140 ± 0.0009	0.0144 ± 0.0001	0.0149 ± 0.0001	0.0159 ± 0.0016
6	0.0275 ± 0.0029	0.0373 ± 0.0034	0.0442 ± 0.0034	0.0545 ± 0.0051	0.0632 ± 0.0066	0.0710 ± 0.0074	0.0818 ± 0.0089	0.0885 ± 0.0099	0.1003 ± 0.0145
	0.0013 ± 0.0001	0.0015 ± 0.0001	0.0017 ± 0.0000	0.0019 ± 0.0001	0.0020 ± 0.0000	0.0022 ± 0.0000	0.0025 ± 0.0001	0.0026 ± 0.0001	0.0028 ± 0.0002
	0.0117 ± 0.0002	0.0123 ± 0.0001	0.0134 ± 0.0009	0.0139 ± 0.0000	0.0147 ± 0.0001	0.0157 ± 0.0006	0.0163 ± 0.0001	0.0173 ± 0.0007	0.0179 ± 0.0002
7	0.0512 ± 0.0058	0.0637 ± 0.0035	0.0779 ± 0.0050	0.0894 ± 0.0099	0.1004 ± 0.0077	0.1118 ± 0.0114	0.1236 ± 0.0100	0.1432 ± 0.0161	0.1494 ± 0.0183
	0.0014 ± 0.0001	0.0016 ± 0.0000	0.0018 ± 0.0001	0.0020 ± 0.0001	0.0023 ± 0.0001	0.0026 ± 0.0008	0.0028 ± 0.0001	0.0029 ± 0.0001	0.0031 ± 0.0001
	0.0126 ± 0.0007	0.0138 ± 0.0006	0.0149 ± 0.0001	0.0165 ± 0.0011	0.0177 ± 0.0006	0.0189 ± 0.0001	0.0217 ± 0.0073	0.0221 ± 0.0007	0.0229 ± 0.0003
8	0.1105 ± 0.0181	0.1287 ± 0.0150	0.1480 ± 0.0147	0.1654 ± 0.0161	0.1814 ± 0.0174	0.1978 ± 0.0150	0.2138 ± 0.0184	0.2273 ± 0.0204	0.2535 ± 0.0258
	0.0017 ± 0.0001	0.0020 ± 0.0001	0.0022 ± 0.0001	0.0025 ± 0.0001	0.0028 ± 0.0001	0.0030 ± 0.0001	0.0034 ± 0.0001	0.0035 ± 0.0001	0.0037 ± 0.0001
	0.0148 ± 0.0031	0.0166 ± 0.0006	0.0189 ± 0.0006	0.0210 ± 0.0007	0.0242 ± 0.0061	0.0264 ± 0.0002	0.0283 ± 0.0008	0.0306 ± 0.0008	0.0335 ± 0.0009
9	0.2391 ± 0.0329	0.2623 ± 0.0333	0.2844 ± 0.0312	0.3099 ± 0.0339	0.3174 ± 0.0239	0.3575 ± 0.0344	0.3780 ± 0.0372	0.4051 ± 0.0465	0.4250 ± 0.0443
	0.0021 ± 0.0000	0.0025 ± 0.0000	0.0029 ± 0.0000	0.0033 ± 0.0010	0.0034 ± 0.0001	0.0038 ± 0.0006	0.0042 ± 0.0001	0.0042 ± 0.0001	0.0045 ± 0.0001
	0.0182 ± 0.0008	0.0224 ± 0.0006	0.0264 ± 0.0002	0.0347 ± 0.0048	0.0355 ± 0.0007	0.0420 ± 0.0011	0.0459 ± 0.0015	0.0525 ± 0.0010	0.0526 ± 0.0014
10	1.4770 ± 0.0517	1.5096 ± 0.0395	1.5268 ± 0.1147	1.5693 ± 0.1020	1.6329 ± 0.0709	1.7133 ± 0.1059	1.7130 ± 0.0922	1.7681 ± 0.1253	1.7853 ± 0.2142
	0.0034 ± 0.0001	0.0039 ± 0.0001	0.0045 ± 0.0007	0.0050 ± 0.0024	0.0049 ± 0.0001	0.0053 ± 0.0002	0.0057 ± 0.0001	0.0057 ± 0.0001	0.0060 ± 0.0002
	0.0267 ± 0.0007	0.0355 ± 0.0010	0.0442 ± 0.0008	0.0529 ± 0.0013	0.0627 ± 0.0016	0.0726 ± 0.0013	0.0817 ± 0.0033	0.0905 ± 0.0025	0.0984 ± 0.0021

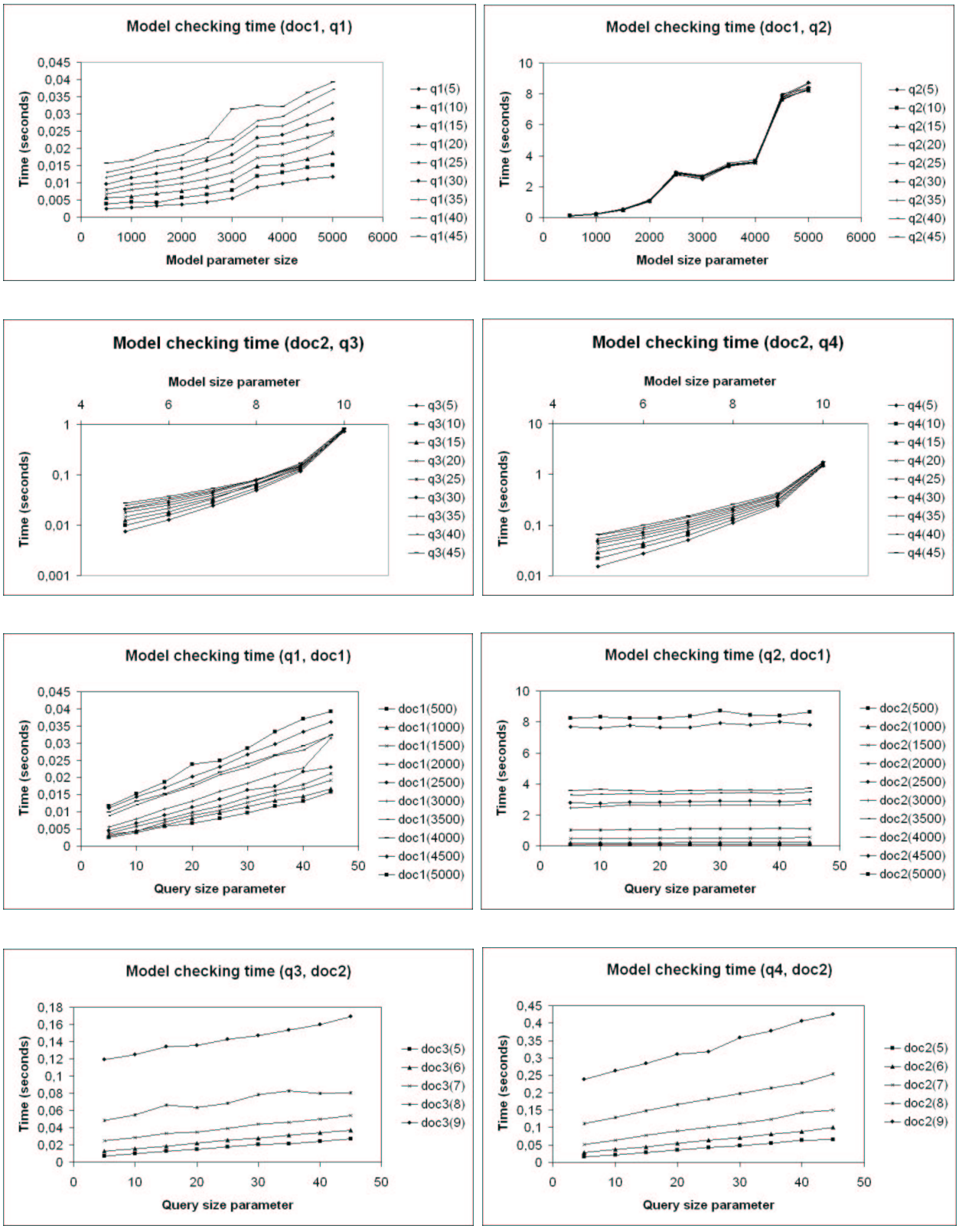


Figure 4.6: Model checking time

Chapter 5

Digestives

In this thesis, we presented a new method for XML query evaluation, based on a reduction to CTL model checking. We provided both the theoretical basis, and a prototype implementation of our idea.

Our approach is well supported from a theoretical point of view. We have devised a linear time translation of the query evaluation problem for Core XPath into the model checking problem for CTL, such that the answer set of a query corresponds to the truth set of its translation. We also have a linear time translation from $\mathcal{X}$CPath to CTL, satisfying the same condition. The translations are based on a reduction from many dimensional CTL to one dimensional CTL.

We implemented the translation, using NuSMV as a model checker for CTL. We ran several experiments to test how well it performs. It turns out that there are practical problems that must be solved in order for the approach to be of practical use. Our experimental results show that the implementation fails to take advantage of the qualities of NuSMV.

Speculations on how to improve our approach

The main problem of our approach is the model representation for NuSMV. First, the compilation time of XCheck, that is, the time taken by NuSMV to translate the document into its internal, OBDD-based representation, is high. The cause of this disappointing behavior is not NuSMV itself, but might be the unconventional way in which we used NuSMV. NuSMV is a *symbolic* model checker. The symbolic representation of a model is usually much smaller than its explicit representation, and in some cases its size is logarithmic in the size of the explicit state model. On the contrary, we provided NuSMV with explicit XML trees whose size is proportional to the size of the XML file. The encoding of such a tree into OBDD format takes a large amount of time, and it is currently not manageable by NuSMV in case of relatively large XML documents.

Another problem is that the pure model checking time, without considering the time spent to encode the model into OBDD format, is still too high compared with the best implementations currently available. We already indicated some reasons for this. The first one is that both the translated model and the translated formula are linear but still bigger than the original XML file and query, respectively. Because of the overhead introduced by the translation of an XML file, we lost part of the regularity that is implicit in most XML documents, and consequently NuSMV failed to compact the translated model into a significantly smaller OBDD format. Moreover, the model checking process uses a pure bottom-up strategy that computes sometimes useless intermediate results. In particular, a bottom-up model checker computes the truth set of all the subformulas of the formula to check.

For a competitive system, at least two improvements have to be made. The first one is to present a much more compact representation of the XML tree to NuSMV. To use the techniques from [8] seem promising. The second one concerns the translation. This should be optimized, keeping in mind the way NuSMV checks its models. Especially for absolute queries (which are evaluated at the root), much better translations are possible.

Incidentally, CTL is not the only logic that we can translate Core XPath into. In fact, recall from Section 2.3 that the fragment of CTL that we use for our translation (i.e., the fragment of CTL without **AU** operators), lies within the expressive bounds of Propositional Dynamic Logic (PDL). An alternative would be to translate Core XPath into PDL and invoke a model checker for PDL. Unfortunately, it seems that the existing PDL model checkers are not at a level at which they can compete with CTL model checkers such as NuSMV.

Finally, what we need in order to cope with realistic problems is a way to store a potentially vast XML file in secondary storage, compress it avoiding redundant parts, load parts of it into main memory, perform some main memory processing and write back the results onto the disk. This is really far from what a model checker does, but we think this is worth investigating.

Back to full XPath

The results reported here only apply to Core XPath and to its extension $\mathcal{X}$CPath. The aim was to see if the general idea is feasible and useful. We would ultimately like to use the same techniques for building an efficient query evaluation system for full XPath. For this, a number of things must be done.

Firstly, we should extend the language in order to be able to reason about attribute information, including `id` attributes, cf. Section 2.2. We do not think that this will pose specific problems, since Definition 2.1.1 is general enough to incorporate such information, if attribute value pairs are considered to be elements of the set of node labels Σ.

Secondly, we will have to add the string and arithmetic operations that come with full XPath. For a general overview of results concerning extending modal logics with arithmetic or string operations, cf. [24]. If we consider our approach feasible and want to extend it to full XPath, this will require more work, since we will have to integrate the process of model checking with the evaluation of string and/or arithmetical expressions.

Bibliography

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 2000.

[2] S. Abiteboul and V. Vianu. Regular path queries with constraints. *Journal of Computer and System Sciences*, 58(3):428–452, 1999.

[3] N. Alechina and M. de Rijke. Describing and querying semistructured data: Some expressiveness results. In S.M. Embury, N.J. Fiddian, W.A. Gray, and A.C. Jones, editors, *Advances in Databases*, LNCS. Springer, 1998.

[4] N. Alechina, S. Demri, and M. de Rijke. A modal perspective on path constraints. *Journal of Logic and Computation*, 2003.

[5] L. De Alfaro. Model checking the world wide web. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification*, pages 337–349. Springer, 2001.

[6] A. Arnold and P. Crubillé. A linear algorithm to solve fixed-point equations on transition systems. *Information Processing Letters*, 29(2):57–66, 1988.

[7] P. Buneman, W. Fan, and S. Weinstein. Path constraints on semistructured and structured data. In *Proceedings PODS '98*, pages 129–138, 1998.

[8] P. Buneman, M. Grohe, and C. Koch. Path queries on compressed XML. In *Proceedings of the International Conference on Very Large Data Bases*, Berlin, 2003.

[9] J. R. Burch, E. M. Clarke, D. E. Long, K. L. MacMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, April 1994.

[10] D. Calvanese, G. De Giacomo, and M. Lenzerini. Representing and reasoning on XML documents: A description logic approach. *Journal of Logic and Computation*, 9(3):295–318, 1999.

[11] R. Cavada, A. Cimatti, E. Olivetti, M. Pistore, and M. Roveri. *NuSMV User Manual*. http://nusmv.irst.itc.it/NuSMV/userman/.

[12] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[13] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model verifier. In *Proceedings of the International Conference on Computer Aided Verification*, volume 1633 of *Lectures Notes in Computer Science*, pages 495–499, 1999.

[14] E. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal-logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[15] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, 1999.

[16] World Wide Web Consortium. Extensible markup language (XML). Available at http://www.w3.org/XML, 1998.

[17] World Wide Web Consortium. XML path language (XPath) version 1.0 – W3C recommendation. Available at http://www.w3.org/TR/xpath.html, 2000.

[18] Mark Jason Dominus. Perl regular expression matching is np-hard. *The Perl Journal*.

[19] G. Gottlob and C. Koch. Monadic Queries over Tree-Structured Data. In *Logic in Computer Science*, pages 189–202, Los Alamitos, CA, USA, July 22–25 2002. IEEE Computer Society.

[20] G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. In *Proceedings of the VLDB Conference*, 2002.

[21] G. Gottlob, C. Koch, and R. Pichler. The complexity of XPath query evaluation. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 179–190, New York, 2003. ACM Press.

[22] J. Y. Halpern, R. Harper, N. Immerman, P. G. Kolaitis, M. Vardi, and V. Vianu. On the unusual effectiveness of logic in computer science. *The Bulletin of Symbolic Logic*, 7(2):213–236, 2001.

[23] C. Koch. Efficient processing of expressive node-selecting queries on XML data in secondary storage: A tree automata-based approach. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2003.

[24] C. Lutz. Description logics with concrete domains—a survey. In *Advances in Modal Logics*, volume 4. World Scientific Publishing Co. Pte. Ltd., 2002.

[25] Maarten Marx. Xpath with conditional axes. Manuscript.

[26] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell Massachusetts, 1993.

[27] G. Miklau and D. Suciu. Containment and equivalence for an XPath fragment. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 65–76, 2002.

[28] E. Quintarelli. *Model-Checking Based Data Retrieval: an application to semistructured and temporal data*. PhD thesis, Dipartimento di Elettronica e Informazione, Politecnico di Milano, 2000.

[29] Michel Rodriguez. XML::Twig - a perl module for processing huge XML documents in tree mode (module version: 3.11). Available at `http://www.xmltwig.com/xmltwig/`, 2000.

[30] Philippe Schnoebelen. The complexity of temporal logic model checking. In *Advances in Modal Logic*, volume 4, pages 437–459. King's College Publications, 2003.

[31] J. Shanmugasundaram, H. Gang, K. Tufte, C. Zhang, D. J. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 302–314. Morgan Kaufmann, 1999.

Appendix A

Source code

```
#!/usr/bin/perl5 -w

use Getopt::Long;
use Time::HiRes qw(gettimeofday tv_interval);
use XML::Twig;
use globals qw($inverse $axe_a $axe_t $α $root $atroot $smv_template);
require "globals";

#------------------
# globals and constants

# query translation
    $inverse  =  { 'self'                  ⇒  'self',
                   'child'                 ⇒  'parent',
                   'parent'                ⇒  'child',
                   'descendant'            ⇒  'ancestor',
                   'ancestor'              ⇒  'descendant',
                   'descendant_or_self'    ⇒  'ancestor_or_self',
                   'ancestor_or_self'      ⇒  'descendant_or_self',
                   'following_sibling'     ⇒  'preceding_sibling',
                   'preceding_sibling'     ⇒  'following_sibling',
                   'following'             ⇒  'preceding',
                   'preceding'             ⇒  'following'
                 };
    $α        =  'EX (d = _a_ & \α)';
    $root     =  '! EX (d = _up_ & EX d = _up_)';
    $atroot   =  "EX E[ d = _up_ U d = _up_ & EX ($root & d = _a_ & \α) ]";
    $axe_a    =  { 'self'                  ⇒  '\α',
                   'child'                 ⇒  "EX (d = _down_ & EX (d = _down_ & $α))",
                   'parent'                ⇒  "EX (d = _up_ & EX (d = _up_ & $α))",
                   'descendant'            ⇒  "EX (d = _down_ & EX E[ d = _down_ U d = _down_ & $α ])",
                   'ancestor'              ⇒  "EX (d = _up_ & EX E[ d = _up_ U d = _up_ & $α ])",
                   'descendant_or_self'    ⇒  "EX E[ d = _down_ U d = _down_ & $α ]",
                   'ancestor_or_self'      ⇒  "EX E[ d = _up_ U d = _up_ & $α ]",
                   'following_sibling'     ⇒  "EX (d = _right_ & EX E[ d = _right_ U d = _right_ & $α ])",
                   'preceding_sibling'     ⇒  "EX (d = _left_ & EX E[ d = _left_ U d = _left_ & $α ])",
                   'following'             ⇒  "EX E[ d = _up_ U d = _up_ & EX (d = _a_ &EX (d = _right_ & EX
 E[ d = _right_ U d = _right_ & EX (d = _a_ & EX E[ d = _down_ U d = _down_ & $α ]) ]) ]",
                   'preceding'             ⇒  "EX E[ d = _up_ U d = _up_ & EX(d = _a_ & EX (d = _left_ & EX
E[ d = _left_ U d = _left_ & EX (d = _a_ & EX E[ d = _down_ U d = _down_ & $α ]) ]) ]"
```

```
                };

    $axe_t     =  { 'self'                    =>  '',  # can't be ommited
                    'child'                   =>  'EX (d = _down_ & EX d = _down_)',
                    'parent'                  =>  'EX (d = _up_ & EX d = _up_)',
                    'descendant'              =>  'EX (d = _down_ & EX d = _down_)',
                    'ancestor'                =>  'EX (d = _up_ & EX d = _up_)',
                    'descendant_or_self'      =>  '',  # can't be ommited
                    'ancestor_or_self'        =>  '',  # can't be ommited
                    'following_sibling'       =>  'EX (d = _right_ & EX d = _right_)',
                    'preceding_sibling'       =>  'EX (d = _left_ & EX d = _left_)',
                    'following'               =>  'EX E[ d = _up_ U d = _up_ & EX (d = _a_ &
EX (d = _right_ & EX d = _right_)) ]',
                    'preceding'               =>  'EX E[ d = _up_ U d = _up_ & EX (d = _a_ &
EX (d = _left_ & EX d = _left_)) ]'
                };

 $smv_template =  q!
MODULE main
VAR s : < VAR > ;
d : {_up_, _down_, _left_, _right_, _a_};
labels : { < LABELS > };

TRANS
 < TRANS >

DEFINE
l : =
case
 < DEFINE >
esac;
!;

 # model translation
 my $model  =  '';
 my $query  =  '';
 my ($t0,$t1,$mtt,$qtt,$rtt);
 my $id  =  0;                                  # an auxiliar variable for indexing the tags
 my %lab_list;
 my $trans_list;
 my $stat;

sub translate_model{
#-------------------------------------------------------------------------
# XML processing using Twig module

 my $model  =  shift;
 my $smv;
    $id = 0;
    %lab_list  =  ();
    $trans_list  =  '';

    $t0  =  [gettimeofday]; # processesing CPU elapse time for model translastion

# parse the XML using XML:: Twig module
 my $tree  =  new XML:: Twig(
```

```
# set the id for each tag except #PCDATA
 StartTagHandlers  ⇒  { '_all_'  ⇒  \&set_id },

# prepare the data to be droped to smv file
TwigHandlers        ⇒  { '_all_'  ⇒  \&to_smvformat }
);

   $tree→parse( $model );                          # process the twig

   $t1  =  [gettimeofday]; # /processesing CPU elapse time for model translastion
   $mtt  =  tv_interval $t0, $t1; # CPU elapse time for model translastion


   $smv  =  parsesmv($id−1,\%lab_list,$trans_list);
   $tree→purge;

   return $smv;
} # \sub translate model

 sub set_id{
 #-------------------------------------------
 # set unique object identifier to each tag entry as a new attr id = 'id'
      unless( $_[1]→gi eq "#PCDATA" ) { $_[1]→set_id($id++) }

 }# \sub

 sub to_smvformat{
 #-------------------------------------------
 # prepare the data to be droped to smv file
   my ($tree, $node)  =  @_;                  # the twig and the current element
      unless( $node→gi eq "#PCDATA" ){
        my $id  =  $node→id;       # unique object identifier
        my $tag  =  $node→gi;      # simple tag

            # define VAR, labels, DEFINE
            $lab_list{$tag} .  =   $lab_list{$tag} ? ",$id" : $id;

            # proceed with TRAN
            $trans_list .  =  "( s  =  $id & d  =  _a_ → (next(s)  =  $id & next(d)  ≠  _a_) ) &\n";

            if( $node→parent ){ # get its parent
            my $parent_id  =  $node→parent→id;
                $trans_list .  =  "( s  =  $id & d  =  _up_ → ( (next(s)  =  $id
            & next(d)  =  _a_) | (next(s)  =  $parent_id & next(d)  =  _up_) ) ) &\n";
            }else{
                $trans_list .  =  "( s  =  $id & d  =  _up_ → (next(s)  =  $id & next(d)  =  _a_) ) &\n";
            }

            if( $node→prev_sibling ){ # prev_sibling & prev_sibling∧−1
            my $prev_sibling_id  =  $node→prev_sibling→id;
                $trans_list .  =  "( s  =  $id & d  =  _left_ →( (next(s)  =
            $id & next(d)  =  _a_) | ( next(s)  =  $prev_sibling_id & next(d)  =  _left_) ) ) &\n";
                # next_sibling for previous one (i.e. prev_sibling∧−1)
                $trans_list .  =  "( s  =  $prev_sibling_id & d  =  _right_ →
            ( (next(s)  =  $prev_sibling_id & next(d)  =  _a_) | (next(s)  =  $id &  next(d)  =  _right_) ) )&\n";
            }else{
                $trans_list .  =  "( s  =  $id & d  =  _left_ → (next(s)  =  $id & next(d)  =  _a_) ) &\n";
            }
```

```
        my @children = $node→children;
           if( @children ){ # children
                @children = map{ $_ = $_→id ? $_→id : "undef"} @children;
           my $last_child_id = $children[$#children];
                $trans_list . = "( s = $id & d = _down_ → ( (next(s) = $id & next(d) = _a_) ";

             while( @children ){
                 my $child_id = shift @children;

                 unless( $child_id eq "undef" ){
                   if( $child_id ne $last_child_id ){
                         $trans_list . = "| (next(s) = $child_id & next(d) = _down_) ";

                   }else{# last_child_id
                              if( !$id ){
                                  $trans_list . = "| (next(s) = $child_id & next(d) = _down_) ) ) &\n";
                                  $trans_list . = "( s = $last_child_id & d = _right_ → (next(s) =
$last_child_id & next(d) = _a_) ) &\n";
                                  $trans_list . = "( s = $id & d = _right_ → (next(s) = $id & next(d) = _a_) );";
                              }else{
                                  $trans_list . = "| (next(s) = $child_id & next(d) = _down_) ) ) & \n";
                                  $trans_list . = "( s = $last_child_id & d = _right_ → (next(s) =
$last_child_id & next(d) = _a_) ) &\n";
                              }
                   }# /if not last_child
                 } # unless

             }# /while children

           }else{ # if @children
             if( !$id ){
                 $trans_list . = "( s = $id & d = _down_ → (next(s) = $id & next(d) = _a_) );\n";
              }else{
                $trans_list . = "( s = $id & d = _down_ → (next(s) = $id & next(d) = _a_) ) &\n";
              }
           }

        }else{
         $node→delete;
        }# \unless

    }# \sub

    sub parsesmv{
    #-------------------------------------------------------
    # parse the smv template
     my ($id,$lab,$trans) = @_;
     my $fsm = $smv_template;

     my $labels = join ',', keys %$lab;
     my $define = ";

     foreach my $l (keys %$lab){
            if( $lab→{$l} = ~ m/,/ ){
                $define . = "s in {$lab→{$l}} : $l;\n";
            }else{
                $define . = "s = $lab→{$l} : $l;\n";
            }
      }

     if( $id ){
       $fsm = ~ s/ < VAR > /0\.\.$id/g;
     }else{
       $fsm = ~ s/ < VAR > /\{0\}/g;
     }
      $fsm = ~ s/ < LABELS > /$labels/g;
```

```
    $fsm = ~ s/ < TRANS > /$trans/g;
    $fsm = ~ s/ < DEFINE > /$define/g;

    return $fsm;
    } # \sub parsesmv

sub nusmv{
#-------------------------------------------------------------------------
# run NuSMV
 my ($cmd_file,$output_file) = @_;
 my $status = system("NuSMV -int -load $cmd_file > $output_file");
    die "NuSMV exited funny: $?" unless $status == 0;

} # \sub running nusmv


sub translate_query{
#-------------------------------------------------------------------------
# translates Core XPath query into a CTL formula
 my $cxp = shift;
 my $ppred = {};

# print $cxp;
 # arrange the predicates
 my $tmp = $cxp;
    $tmp = ~ s/( +)/ /g;
    $tmp = ~ s/\[/\{/g;
    $tmp = ~ s/\]/\}/g;
 my $i = 0;

    while( $tmp = ~ s/\{([^\{]*?)\}/\[pred_$i\]/ ){
        $ppred→{"pred_$i"} = $1;
        $i++;
    }
    $cxp = $tmp;

   return τ($cxp,$ppred);

} # \sub translate query

 sub τ{
#-------------------------------------------------------------------------
# query translation
 my ($query,$ppred) = @_;

 if( $query = ~ m/^(.+)\/(.+?):: (.+)\[(.*?)\]$/ ){
                my ($locpath,$tmp,$l,$newpred) = ($1,$axe_a→{$inverse→{$2}},$3,$4);
                my $α = τ($locpath);
                   $tmp = ~ s/\\α/$α/g                    || die "error: invalid query1 $query!\n\n";
                            if($l eq "*"){
                                                    return "d = _a_ & $tmp & ".ω($ppred→{$newpred},$ppred);
                            }else{
                                                    return "d = _a_ & l = $l & $tmp & ".ω($ppred→{$newpred},$ppred);
                            }
 }

 if( $query = ~ m/^(.+)\/(.+?):: (.+)$/ ){
                my ($locpath,$tmp,$l) = ($1,$axe_a→{$inverse→{$2}},$3);
```

```
                my $α = τ($locpath);
                   $tmp = ∼ s/\\α/$α/g                    || die "error: invalid query2 $query!\n\n";
                            if($1 eq "*"){
                                                  return "d = _a_ & $tmp";
                            }else{
                                                  return "d = _a_ & l = $l & $tmp";
                            }
 }

 if( $query = ∼ m/∧\/(.+?):: (.+)\[(.*?)\]$/ ){
                my ($tmp,$1,$newpred) = ($axe_a→{$inverse→{$1}},$2,$3);
                my $α = $root; # root
                   $tmp = ∼ s/\\α/$α/g                                  || die "error: invalid query3 $query!\n\n";
                            if($1 eq "*"){
                                                  return "d = _a_ & $tmp & ".ω($ppred→{$newpred},$ppred);
                            }else{
                                                  return "d = _a_ & l = $l & $tmp & ".ω($ppred→{$newpred},$ppred);
                            }
 }

 if( $query = ∼ m/∧\/(.+?):: (.+)$/ ){
                my ($tmp,$1) = ($axe_a→{$inverse→{$1}},$2);
                my $α = $root; # root
                   $tmp = ∼ s/\\α/$α/g                                  || die "error: invalid query4 $query!\n\n";
                            if($1 eq "*"){
                                                  return "d = _a_ & $tmp";
                            }else{
                                                  return "d = _a_ & l = $l & $tmp";
                            }
 }

 if( $query = ∼ m/∧(.+?):: (.+)\[(.*?)\]$/ ){
                my ($tmp_t,$1,$newpred) = ($axe_t→{$inverse→{$1}},$2,$3);
                my $α = $tmp_t ? "d = _a_ & l = $l & $tmp_t & ".ω($ppred→{$newpred},$ppred) :
"d = _a_ & l = $l & ".ω($ppred→{$newpred},$ppred);
                            if($1 eq "*"){
                                   $α = $tmp_t ? "d = _a_ & $tmp_t & ".ω($ppred→{$newpred},$ppred) :
'd = _a_ & '.ω($ppred→{$newpred},$ppred);
                            }
                   return $α;
 }

 if( $query = ∼ m/∧(.+?):: (.+)$/ ){
                my ($tmp_t,$1) = ($axe_t→{$inverse→{$1}},$2);
                my $α = $tmp_t ? "d = _a_ & l = $l & $tmp_t" : "d = _a_ & l = $l";
                            if($1 eq "*"){
                                   $α = $tmp_t ? "d = _a_ & $tmp_t" : "d = _a_";
                            }
                   return $α;
 }

 die "error: invalid query $query!\n\n";

}

#-------------------------------------------------------------------
 sub ω{
# predicat translation
 my ($pred,$ppred) = @_;
```

```
if( $pred = ~ m/([^\[\]]*) or ([^\[\]]*)/ ){ return "(".ω($1,$ppred)." | ".(ω($2,$ppred))")" }
if( $pred = ~ m/([^\[\]]*) and ([^\[\]]*)/ ){ return ω($1,$ppred)." & ".(ω($2,$ppred)) }
if( $pred = ~ m/^not ([^\[\]]*)/ ){ return " ! (".(ω($1,$ppred))")" }
if( $pred = ~ m/^\/(.+)/ ){
        my $α = ω($1,$ppred);
        my $tmp = $atroot;
           $tmp = ~ s/\\α/$α/g                    || die "error: invalid pred [$pred]!\n\n";
         return $tmp;
}

if( $pred = ~ m/^(.+?):: (.+)\[(.*?)\]\/(.*)$/ ){
        my ($tmp,$l,$newpred,$locpath) = ($axe_a→{$1},$2,$3,$4);
        my $α = "l = $l & ".ω($ppred→{$newpred},$ppred)." & ".ω($locpath,$ppred);
           if($l eq "*"){
                    $α = ω($ppred→{$newpred},$ppred)." & ".ω($locpath,$ppred);
           }
           $tmp = ~ s/\\α/$α/g                    || die "error: invalid pred [$pred]!\n\n";
           return $tmp;
}

if( $pred = ~ m/^(.+?):: (.+?)\/(.*)$/ ){
        my ($tmp,$l,$locpath) = ($axe_a→{$1},$2,$3);
        my $α = "l = $l & ".ω($locpath,$ppred);
           if($l eq "*"){
                    $α = ω($locpath,$ppred);
           }
      $tmp = ~ s/\\α/$α/g                         || die "error: invalid pred [$pred]!\n\n";
           return $tmp;
}

if( $pred = ~ m/^(.+?):: (.+)\[(.*)\]$/ ){
        my ($tmp,$l,$newpred) = ($axe_a→{$1},$2,$3);
        my $α = "l = $l & ".ω($ppred→{$newpred},$ppred);
           if($l eq "*"){
                    $α = ω($ppred→{$newpred},$ppred);
           }
           $tmp = ~ s/\\α/$α/g                    || die "error: invalid pred [$pred]!\n\n";
           return $tmp;
}

if( $pred = ~ m/^(.+?):: (.+)$/ ){
        my ($tmp_t,$tmp,$l) = ($axe_t→{$1},$axe_a→{$1},$2);
           if($l eq "*"){
                    return 'd = _a_ & '.$tmp_t          || die "error: invalid pred [$pred]!\n\n";
           }else{
                    $tmp = ~ s/\\α/l = $l/g             || die "error: invalid pred [$pred]!\n\n";
                    return $tmp;
           }
}

die "error: invalid pred [$pred]!\n\n";
}

sub retrieve{
#-------------------------------------------------------------------
# retrieves the XML elements corresponding with the truth set of the
# translated formula
my ($model,$res_file) = @_;
my %res;
```

```
open STS, "$res_file.mc"                    or die "can't open: $res_file!";
while( my $line =  < STS > ){
   if( $line = ~ /State/){
      $line =  < STS > ; # l = C
   my $id = (split / = /, < STS > )[1]; # s = 3
      $id = ~ m/(\d+)/;
      $id = $1;
      $res{$id} = "";
      $line =  < STS > ; # d = _a_
      $line =  < STS > ; # labels = B
   }
}
close STS;


#-------------------------------------------------------------------------
# parse the XML using XML:: Twig module

my $id = 0;                                          # an auxiliar variable for indexing the tags
my $tree = new XML:: Twig(
       StartTagHandlers => {
# set the id for each tag except #PCDATA
         '_all_' => sub{ unless( $_[1]→gi eq "#PCDATA" ) { $_[1]→set_id($id++) } } } );
      $tree→parse( $model );                         # process the twig

#-------------------------------------------------------------------------
# print the retrieved nodes
    $t0 = [gettimeofday]; # processesing CPU elapse time for model translastion

          $qtt = tv_interval $t0, $t1; # CPU elapse time for model translastion

    my $num = 1;
    open QERES, " > $res_file.qe"                  or die "can't open: $res_file!";
    foreach my $id (sort keys %res){
      my $node = $tree→elt_id($id);
        print QERES " < RES\t$num > \n";
        print QERES $node→sprint."\n"; # parameter 1 − print only the #PDATA, skip the begin and end tag
        print QERES " < /RES > \n\n";
        $num++;
    }
    close QERES;
    $t1 = [gettimeofday]; # /processesing CPU elapse time for retrieving the results
    $rtt = tv_interval $t0, $t1; # CPU elapse time for model translastion

    $tree→purge;
} # \sub retrieve
```