PROCEEDINGS

AUTOMATED SPECIFICATION AND VERIFICATION OF WEB SYSTEMS

WWV 2010

SIXTH INTERNATIONAL WORKSHOP
JULY 30-31, 2010
VIENNA, AUSTRIA

LAURA KOVÁCS AND TEMUR KUTSIA (Eds.)
Preface


WWV’10 provided a common forum for researchers from the communities of Rule-based Programming, Automated Software Engineering, and Web-oriented research, in order to facilitate the cross-fertilization and the advancement of hybrid methods that combine the three areas.

The Program Committee of WWV’10 collected three reviews for each paper and held an electronic discussion in May 2010 which has led to the selection of 5 regular papers. In addition to the selected papers, the scientific program included two invited talks by Thomas Eiter from Vienna University of Technology (Austria) and Andrei Voronkov from the University of Manchester (UK). We would like to thank them for having accepted our invitation.

We would also like to thank all the members of the Program Committee and all the referees for their careful work in the review and selection process. Many thanks to all authors who submitted papers and to all conference participants.

We thank Andrei Voronkov for his help on EasyChair.

Finally, we gratefully acknowledge Vienna University of Technology who supported this event.

Vienna
July 2010
Laura Kovács and Temur Kutsia
WWV’10 Program Chairs
Workshop Organization

Program Committee

María Alpuente  Technical University of Valencia, Spain
Demis Ballis  University of Udine, Italy
Jorge Manuel Neves Coelho  University of Porto, Portugal
Vicente Luque Centeno  Universidad Carlos III de Madrid, Spain
Wlodzimierz Drabent  IPI PAN, Polish Academy of Sciences, Poland
Santiago Escobar  Technical University of Valencia, Spain
Moreno Falaschi  University of Siena, Italy
Jens Knoop  Vienna University of Technology, Austria
Laura Kovács  Vienna University of Technology, Austria (chair)
Temur Kutsia  Johannes Kepler University Linz, Austria (chair)
Catherine Meadows  Naval Research Laboratory, USA

Local Organization

Laura Kovács

External Reviewers

Linda Brodo
Daniel Romero

Sponsoring Institutions

Vienna University of Technology
# Table of Contents

Answer Set Programming in a Nutshell *(invited talk)*  ............................................ 1  
*Thomas Eiter*

EasyChair *(invited talk)*  .......................................................... 2  
*Andrei Voronkov*

Development of a Query Language for GML based on XPath  .......................................... 3  
*Jesus Almendros-Jimenez, Antonio Becerra-Teron, Francisco Garcia-Garcia*

Temporal Patterns for Document Verification  ................................................................. 17  
*Mirjana Jaksic, Burkhard Freitag*

Specifying Hyperdocuments with Algebraic Methods  ...................................................... 32  
*Volker Mattick*

Reconstructing Information Retrieved from Multiple Websites  ........................................ 46  
*Héctor Valero, Carlos Castillo, Josep Silva*

Incremental Construction of Counterexamples in Model Checking Web Documents ............... 61  
*Franz Weitl, Shin Nakajima*
Answer Set Programming in a Nutshell

- Invited Talk -

Thomas Eiter
Institut für Informationssysteme, Technische Universität Wien
eiter@kr.tuwien.ac.at

Abstract

Answer Set Programming (ASP) has emerged in the recent years as a powerful paradigm for declarative problem solving, which has its roots in knowledge representation and non-monotonic logic programming. Similar to SAT solving, the basic idea is to encode solutions to a problem in the models of a non-monotonic logic program, which can be computed by reasoning engines off the shelf. ASP is particularly well-suited for modeling and solving problems which involve common sense reasoning or transitive closure, and has been fruitfully applied to a growing range of applications. Among the latter are also problems in testing and verification, for which efficient core fragments of ASP that embrace Datalog haven been exploited. This talk gives a brief introduction to ASP, covering the basic concepts, some of its properties and features, and solvers. It further addresses some applications in the context of verification and recent developments in ASP, which bring evaluation closer to other formalisms and logics.
EasyChair

- Invited Talk -

Andrei Voronkov
The University of Manchester
andrei@voronkov.com

Abstract

EasyChair is the most commonly used conference management system. Currently about twelve conferences or workshops per day register for using EasyChair. There are about 1,200 papers submitted to EasyChair every day. The number of users of EasyChair at the time of writing this abstract is over 300,000, which is greater than the population of Linz.

In this talk we give an overview of EasyChair and describe its philosophy, design, implementation, evolution and future. We will also discuss issues related to formal analysis and verification of Web services.
Development of a Query Language for GML based on XPath*

Jesús M. Almendros-Jiménez
University of Almería
Almería, Spain
jalmen@ual.es

Antonio Becerra-Terón
University of Almería
Almería, Spain
abecerra@ual.es

Francisco García-García
University of Almería
Almería, Spain
paco.garcia@ual.es

Abstract

Geography Markup Language (GML) has been established as the standard language for the transport, storage and modelling of geographic information. In this paper we study how to adapt the XPath query language to GML documents. With this aim, we have defined a semantic based XPath language which is not based on the (tree-based) syntactic structure of GML documents, instead it is based on the “semantic structure” of GML documents. In other words, the proposed XPath language is based on the GML schema. We have developed a system called UALGIS, in order to implement the approach. Such system stores GML documents by means of the PostGIS RDBMS. In order to execute semantic-based XPath queries we have defined a translation of the queries into SQL. Such translation takes into account the GML schema. Finally, the system allows to visualize the result. With this aim, the result of a query is exported to the Keyhole Markup Language (KML) format.

1 Introduction

The Geography Markup Language (GML) [7, 20, 21, 8] has been established as the standard language for the transport, storage and modeling of geographic information. GML is a dialect of the eXtensible Markup Language (XML) [27], which adapts XML to Geo-spatial data. XML allows to describe the structure of Web data by means of a tree. The tree structure is used to describe relations between data: for instance, a paper tag contains author, title and publisher as subtree tags and the subtree publisher can be described by means of name of the journal, country, editors, etc. The need for querying XML documents has motivated the design of the XPath query language [28]. The XPath language allows to specify the path of the XML tree to be retrieved. In addition, XPath allows to constraint the query by means of boolean conditions about the attributes and tags of the selected nodes. For instance, we can specify in a query that we would like to retrieve the editors of the journals in which “Becerra” has published a paper as follows:

```
/papers/paper[author = “Becerra”]/journal/editor
```

XPath can be used for retrieving the GML elements. However, due to the usual syntactic structure of GML documents XPath should be adapted to this special case. GML allows to describe spatial objects including, if any, their geometry, together with the coordinate reference system, topology, among others. However, usually, GML documents do not use the tree-based structure of XML documents for the representation of spatial objects. Usually, GML documents store spatial objects as a sequence of XML elements, and they are stored as children of the tree root [18, 30]. It makes the tree-based XPath useless in most of the cases. The reason for storing spatial objects as a sequence of children of the root is that the tree structure is not used for representing spatial relations. One could think that a subtree of a node represents, for instance, the spatial objects enveloped by the node. However, it is not true in general. GML allows the specification of relations between spatial objects. However, GML allows to define a vocabulary of relations between spatial objects. For instance, the European INSPIRE Directive [12] has defined a certain vocabulary of GML whose aim is to create a spatial data infrastructure in the European

*This work has been partially supported by the Spanish MICINN under grant TIN2008-06622-C03-03.
Development of a query language for GML based on XPath

Almendros, Becerra, and García

Union in order to share information through public organizations. However, the syntactic structure of GML documents do not necessarily take into account the “semantic structure” of such vocabulary. Such vocabulary can be seen as a GML schema in such a way that spatial objects and relations between them conforms the schema. Usually, GML documents contain spatial objects in which spatial relations are specified by means of the linking mechanism of XML documents. Using XPath for GML documents, we could follow the links of the GML document in order to retrieve relationships between spatial objects, however, it makes XPath queries very sophisticated.

In this paper we study how to adapt the XPath query language to GML documents. With this aim, we have defined a semantic based XPath language which is not based on the (tree-based) syntactic structure of GML documents, rather than, it is based on the “semantic structure” of GML documents. In other words, the proposed XPath language is based on the GML schema instead of the syntactic structure.

We have developed a system called UALGIS, available via Web in http://indalog.ual.es/ualgis/testGMaps.jsp in order to implement the approach. Such system stores GML documents by means of the PostGIS [25]. In order to execute semantic-based XPath queries we have defined a translation of the queries into SQL. Such translation takes into account the GML schema. Finally, the system allows to visualize the result. With this aim, the result of a query is exported to the Keyhole Markup Language (KML) format [6].

1.1 Related Work

Spatial data can be handled by well-known relational database management systems (RDBMS) like: SpatialSQL [11], GeoSQL [13], Oracle Spatial [26] and PostGIS [25]. Basically, they are based on extensions of the relational model for storing spatial objects and extensions of the SQL query language for the retrieval of spatial queries.

In the case of GML data, the Web Feature Service (WFS) is a standard of the OpenGis Consortium (OGC) [22] for data manipulation of geographic features stored on a Web site (i.e. a Web Feature Server) using HTTP requests. The expressiveness of this language is very poor compared with query languages like SQL. GQuery [4] is a proposal for adding spatial operators to XQuery [5], the standard XML query language. Manipulation of trees and sub-trees are carried out by means of XQuery, while spatial processing is performed using geometric functions which use the JTS Topology Suite [1]. JTS is an open source API that provides a spatial object model and a set of spatial operators. The GeoXQuery approach [14] extends the Saxon XQuery processor [16] with function libraries that provide geo-spatial operations. It is also based on JTS and provides a GML to Scalable Vector Graphics (SVG) [29] transformation library for the XQuery processor in order to show query results.

With respect to GQuery and GeoXQuery, our proposal can be seen as a specific query language for GML instead of considering ad-hoc mechanisms for querying GML in XPath and XQuery. Our approach is focused on the XPath query language which is a sublanguage of the XQuery language. Our work can be seen as the first step to use XQuery as query language for GML documents. However, our proposal is based on the semantic structure of GML documents instead of the syntactic one used in GQuery and GeoXQuery. With respect to geometric functions, we are not still interested in queries involving geometric operations. Our current approach is mainly interested to querying semantic spatial relations expressed in GML documents which do not depend on the geometry of the objects. In any case, we will consider the extension of our work to geometric operations in the future by using the PostGIS library. Finally, our system is able to export the result of queries in order to visualize them, but instead of using SVG like in GeoXQuery, we export to KML.

GML Query [19] is also a contribution in this research line that stores GML documents in a spatial RDBMS. This approach performs a simplification of the GML schema that is then mapped to its corresponding relational schema. The basic values of spatial objects are stored as values of the tables. Once
the document is stored, spatial queries can be expressed using the XQuery language with spatial functions. The queries are translated to their equivalent in SQL which are executed by means of the spatial RDBMS. This approach has some similarities with our. Firstly, the storage of GML documents in a spatial RDBMS. In our system, we store GML documents by means of the PostGIS RDBMS. Secondly, in our approach the queries are expressed in XPath and are translated into SQL. However, our XPath-based query language is properly based on the GML schema.

Another problem related to GML is how to visualize GML documents. There are several technologies (i.e., SVG, VRML, HTML, among others) to specify how to show the content of GML documents. KML [6] is an XML-based language focused on geographic data visualization, including annotation of maps and images, as well as controlling the display in the sense of where to go and where to look. From this perspective, KML is complementary to GML and most of the major standards of the OGC including Web Feature Service (WFS) [23] and Web Map Service (WMS) [24]. We have decided to export our GML query result to KML due to the advantages that offer this technology (i.e., APIs, WFS, WMS) and because it has been approved by the OGC as a standard for the exchange and representation of geographic data in three dimensions. In this way, the results can be interpreted by different GIS or Earth browsers. Other approaches return the output in SVG format but it has as disadvantage that it is just a graphical format. KML can display the results without losing the GML semantics and allows to include meta-data.

The rest of this article is organized as follows. Section 2 will present the GML data model and schema. Section 3 will define the semantic-based XPath language. Section 4 will describe the system and finally Section 5 will conclude and present future work.

2 GML Data Model

Next, we show an example of GML document representing the center of a city:

```xml
<CityCenter gml:id="C1">
   <gml:name>London</gml:name>
   <geometry>
      <gml:Point>
         <gml:pos>45.256 -71.92</gml:pos>
      </gml:Point>
   </geometry>
   <cityCenterMember>
      <Building gml:id="B1">
         <gml:name>Great Building</gml:name>
         <belongsTo>
            <Block xlink:href="BL1"/>
         </belongsTo>
      </Building>
   </cityCenterMember>
   <cityCenterMember>
      <Building gml:id="B2">
         <gml:name>Small Building</gml:name>
         <belongsTo>
            <Block xlink:href="BL2"/>
         </belongsTo>
      </Building>
   </cityCenterMember>
</CityCenter>
```
As above mentioned, this GML document describes a center of a city. This description includes tags for \textit{CityCenter} and \textit{cityCenterMember}'s of a city, such as \textit{buildings}, \textit{blocks} and \textit{ways}. However, a GML data model provides mechanisms for structuring GML documents. For instance, the previous GML document can be also represented as follows:

\begin{verbatim}
<cityCenterMember>
  <Block gml:id="BL1">
    <gml:name>Grey Block <gml:/name>
    <nextTo>
      <Way xlink:href="W1"/>
      <Way xlink:href="W2"/>
    </nextTo>
  </Block>
<cityCenterMember>
  <Block gml:id="BL2">
    <gml:name>Green Block <gml:/name>
  </Block>
<cityCenterMember>
  <Way gml:id="W1">
    <gml:name>6th Street <gml:/name>
  </Way>
<cityCenterMember>
  <Way gml:id="W2">
    <gml:name>5th Street <gml:/name>
  </Way>
<cityCenterMember>
</CityCenter>
\end{verbatim}
where instead of using linking mechanisms of XML (i.e. xlink:href), the elements of the GML document are nested. However, it is normally recommended to avoid nesting of GML elements in order to not increase the complexity of GML documents. From the point of view of using XPath for querying GML documents, it makes not easy to express complex queries w.r.t. GML documents. Our approach aims to propose a semantic version of XPath in which the result of the query does not depend on the syntactic structure of the GML document. In other words, nesting of elements is not relevant in our approach because we will follow the GML schema to define queries. Next will present a standard of GML schemas to be used in our approach.

2.1 INSPIRE Directive

The European INSPIRE Directive [12] aims to create a spatial data infrastructure in the European Union to share information through public organizations and facilitate public access across Europe. Furthermore, the spatial information considered under this policy is extensive and covers a wide range of areas and topics. INSPIRE is based on several common principles:

- Data should be collected once and should be stored where they can be maintained more efficiently.
- It must be able to easily combine the spatial information from different sources across Europe and share it with other users and applications.
- It should be possible to collect information at some detail level and share it with all levels, e.g. detailed for local analysis, general for global strategic purposes, . . .
- Geographic information should be transparent and readily available.
- It must be easy to find which geographic information is available, how it can be used to meet a specific need, and under which conditions can be acquired and used.

![GML Schema of the City Center](image)

**Figure 1: GML Schema of the City Center**

The INSPIRE directive defines 34 topics on spatial data needed for application development. The INSPIRE directive defines GML schemas for each one of these topics. We have followed this directive in our approach in order to make our proposal more interesting in the real world.

For instance, the Figure 1 shows an example of GML schema for cities. This schema uses an UML Profile for its definition called HollowWorld [9] which is based on Table E.1 of ISO 19136:2007 (GML 3.2.1) [2] and is used by the INSPIRE Directive. It describes the elements (i.e. the members) of a city center. A city center contains different entities like locals, buildings, blocks, monuments and ways. Locals can be bars, shops and banks. Buildings can be buildings of interest, and ways can be roundabouts, streets and squares. The GML schema also includes spatial relationships between the entities: locals are ‘locatedAt’ buildings that “belongsTo” blocks which can be “nextTo” Ways. Monuments can be also
“locatedAt” a way. GML allows to describe entities by means of the Feature type. In addition, the INSPIRE directive provides Data types for geographical entity naming.

In our system, we have used data available from IDEAndalucía [15] which is part of the geo-services of the INSPIRE directive. This is an Andalusian Cartographic System Geo-portal available to search, locate, view, download or request some type of geographic information referring to the territory of Andalusia in Spain. IDEAndalucía provides various services, such as WMS (Web Map Server) and WFS (Web Feature Service) with data available in GML format. By a GetFeature request to a WFS server a GML document is returned with all the Features of a selected type that are within the limits defined in a geographical rectangle or GML Bounding Box.

### 3 Development of a Query language for GML based on XPath

Now, the proposed GML query language is as follows. Basically, the path of the query has to follow the GML schema, and the query can include boolean conditions over the elements of the GML schema. For instance, we can express the following queries w.r.t. the running example:

<table>
<thead>
<tr>
<th>Query 1. Buildings of the Block named “Grey Block”</th>
<th>Query 2. Ways called “5th street” next to some Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>/Building[belongsTo/Block/gml:name=&quot;Grey Block&quot;]</td>
<td>/Block/nextTo/Way[gml:name=&quot;5th Street&quot;]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query 3. Ways nextTo a Building called “Great Building”</th>
</tr>
</thead>
<tbody>
<tr>
<td>/Building[gml:name=&quot;Great Building&quot;] /belongsTo/Block/nextTo/Way</td>
</tr>
</tbody>
</table>

The semantic version of the XPath query language is syntactically similar to the tree-based version. However, the semantic version can specify paths starting from any point of the schema (i.e. the root of the XPath expression can be any of the features of the GML document). The XPath expression alternates features with spatial relations. For instance, starting from Building we can build the following (semantic) XPath expression:

```
/Building[gml:name="Great Building"] /belongsTo/Block/nextTo/Way
```

by following the sequence Building, belongsTo, Block, nextTo and Way, where Building, Block are Features and belongsTo and nextTo are relationships among Features.

Finally, let us see the syntactic version of the above Query 1, which is considerably more complex than the proposed semantic one:

```
/CityCenter/cityCenterMember/Building[belongsTo/Block
[@xlink:href="/CityCenter/cityCenterMember/Block[gml:name="Grey Block"]/@gml:id]]
```

### 3.1 Translation XPath to SQL

In order to implement the proposed XPath-based query language, we have to proceed as follows:

1. A GML schema is transformed into a Relational schema.
2. A GML document is stored in the spatial RDBMS.
3. A XPath query is translated into an equivalent SQL query.
4. The result of the query is exported to GML or KML format.

3.1.1 Transforming the GML Schema into a Relational Schema

For data storage the first thing to do is to transform the GML schema into a relational schema of PostGIS. For this transformation we proceed as follows:

- A table is created for each element of Feature type.
- Attributes of elements of Feature type are mapped to columns of the tables.
- Geometric attributes of elements of Feature type are mapped to columns of PostGIS geometry type.
- Spatial relations between elements of Feature type are represented in two ways:
  - A one to one relationship is mapped to a column that references the primary key of the elements in the spatial relation.
  - A one to many relationship is mapped to a table, with the name of the spatial relation, with columns containing the primary keys (i.e. foreign keys) of the elements in the spatial relation.
  - A many to many relationship is mapped as two one-to-many relationships, one for each direction of the relationship.
- Feature inheritance is represented by table inheritance provided by PostGIS.

Figure 2 shows the result of the transformation of some elements of the GML schema represented in Figure 1.

![Figure 2: A Fragment of Relational schema of the City Center](image)

3.1.2 Storage of GML documents in the Spatial RDBMS

The GML documents are stored in the spatial RDBMS as follows. Firstly, features instances are added to tables. Secondly, spatial relations are added to columns (in the case of one to one relationships) and to tables (in the case of one to many and many to many relationships). Next, we show the table instances of the running example:
### Table: Building

<table>
<thead>
<tr>
<th>ogc_fid</th>
<th>name</th>
<th>belongsTo</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>Great Building</td>
<td>BL1</td>
</tr>
<tr>
<td>B2</td>
<td>Small Building</td>
<td>BL2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

### Table: Block

<table>
<thead>
<tr>
<th>ogc_fid</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>BL1</td>
<td>Grey Block</td>
</tr>
<tr>
<td>BL2</td>
<td>Green Block</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

### Table: Way

<table>
<thead>
<tr>
<th>ogc_fid</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>6th Street</td>
</tr>
<tr>
<td>W2</td>
<td>5th Street</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

### Table: Block_nextTo

<table>
<thead>
<tr>
<th>Block</th>
<th>Way</th>
</tr>
</thead>
<tbody>
<tr>
<td>BL1</td>
<td>W1</td>
</tr>
<tr>
<td>BL1</td>
<td>W2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

#### 3.1.3 Translation of XPath into SQL

We can now define the translation of XPath-based queries into SQL queries, using the transformation of the GML Schema into the Relational Schema. The translation is as follows:

1. Case `/A/p/B` where `p` is a one to one relationship: Select `B.* From A,B Where A.p = B.id`


5. Case `/A[cond]/p/B[cond2]` where `p` is a one to one relationship: Select `B.* From A,B Where A.p = B.id and cond and cond2`


7. Similarly, the rest of the cases
Now, we show the translation into SQL expressions of the XPath queries of Section 3 w.r.t. the GML schema represented by Figure 1.

### Query 1. Buildings of the Block named “Grey Block”

<table>
<thead>
<tr>
<th>XPath Query</th>
<th>SQL Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>/Building[belongsto/Block/gml:name=&quot;Grey Block&quot;]</td>
<td>Select Building.* from Building, Block where Building.belongsto = Block.id and Block.name = “Grey Block”</td>
</tr>
</tbody>
</table>

Next, we show the result of the query in GML format:

```xml
<Building gml:id="B1">
  <gml:name>Great Building</gml:name>
  <belongsTo>
    <Block xlink:href="BL1"/>
  </belongsTo>
</Building>
```

### Query 2. Ways called “5th Street” next to some Block

<table>
<thead>
<tr>
<th>XPath Query</th>
<th>SQL Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>/Block/nextTo/Way[gml:name=&quot;5th Street&quot;]</td>
<td>Select Way.* from Way, Block_nextTo, Block where Block_nextTo.Block = Block.id and Block_nextTo.Way = Way.id and Way.name = “5th Street”</td>
</tr>
</tbody>
</table>

### Query 3. Ways nextTo a Building called “Great Building”

<table>
<thead>
<tr>
<th>XPath Query</th>
<th>SQL Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>/Building[gml:name=&quot;Great Building”]/belongsTo/Block/nextTo/Way</td>
<td>Select Way.* from Way, Block_nextTo, (Select Block.* from Building, Block where Building.belongsto = Block.id and Building.name = “Great Building”) Block_0 where Block_nextTo.Block = Block_0.id and Block_nextTo.Way = Way.id</td>
</tr>
</tbody>
</table>

### 3.1.4 Exporting to GML and XML

*PostGIS* natively provides several functions for the conversion of stored geometries to GML and KML formats:

- *AsGML*. Returns the geometry as a GML element. We can choose the spatial reference system.
Development of a query language for GML based on XPath

Almendros, Becerra, and García

- AsKML. It works similarly to AsGML but returning it as a KML geometry. We cannot choose the spatial reference system since it is fixed to WGS84.

These functions are only responsible for generating the geometry in GML / KML format but it is still required the export tables as GML and KML elements. With this aim, PostGIS provides two tables as follows:

- Geometry_columns table: It stores a catalog with the schema names, table names, column names of the geometric data and their spatial reference system.
- Spatial_ref_sys table: It contains a collection of spatial reference systems and stores information about the projections for transforming from one system to another.

4 UALGIS System

For validating the proposed query language a Web Geographic Information System, called UALGIS, has been implemented (available in [http://indalog.ual.es/ualgis/testGMaps.jsp](http://indalog.ual.es/ualgis/testGMaps.jsp)). The main features of the system can be summarized as follows:

- Storage of GML documents with PostGIS.
- GML / KML document creation using dom4j [17].
- Querying of elements of feature type of the database.
- XPath-based querying.
- Google Maps-based client for displaying the result of queries.

The system has been built by using the Java Eclipse Galileo [10] as IDE. For project management, we have used Maven 2 [3], which has been used for handling library dependences required for the construction of the GIS. The system architecture includes PostGIS version 1.4 server for data storage and a Tomcat version 6.0 server that handles the logic and presentation layers of the application. For the presentation layer, pages are programmed using JSP, HTML and AJAX for interacting with the server. Figure 3 shows the architecture.

![Figure 3: UALGIS System Architecture](image)

The FeatureManager component is responsible for the management of Features stored in the database. It is responsible for translating XPath queries into SQL queries. The FeatureManager is also responsible for transforming the database rows to an object model that can be processed by the DocumentManager component. The DocumentManager component is responsible for handling documents in GML/KML format. It is also responsible for transforming the objects returned by FeatureManager onto KML /
GML documents. It also supports the reverse process, i.e. the extraction of features from GML/KML documents. For creating and reading documents the dom4j library is used. This is an Open Source XML Framework for Java that allows reading, writing and navigation of XML documents. It also includes a processing model based on events for large documents or XML streams and includes support for XML Schema validation types. On the client side, the Web GIS is a map browser with support for KML. It is based on the Google Maps API including a fast and efficient 2D browser that provides features like zooming, panning, searching and displaying geographical names and information about geographic entities. Figure 4 shows a snapshot of the UALGIS system.

Figure 4: Snapshot of UALGIS

5 Conclusion and Future Work

In this paper we have studied how to adapt the XPath query language to GML documents. With this aim, we have defined a semantic based XPath language which is not based on the (tree-based) syntactic structure of GML documents, instead it is based on the “semantic structure” of GML documents. We have developed a system called UALGIS, in order to implement the approach. Such system stores GML documents by means of the PostGIS RDBMS. In order to execute semantic-based XPath queries we have defined a translation of the queries into SQL. Such translation takes into account the GML schema. Finally, the system allows to visualize the result. With this aim, the result of a query is exported to the KML format. As future work different techniques as GML filtering, indexing, etc., will be studied in
order to improve the performance of the UALGIS system. On the other hand, we would like to extend our work to the XQuery language. The extension should be also based on the semantic of GML. Finally, we would like to combine our GML query language with ontologies. Its use would improve the kind of queries and answers obtained from GML documents.

References


Temporal Patterns for Document Verification

Mirjana Jakšić*, Burkhard Freitag
University of Passau
Germany
{Mirjana.Jaksic,Burkhard.Freitag}@uni-passau.de

Abstract

In this paper we present a novel user-friendly high-level approach to the specification of temporal properties of web documents which can be used for verification purposes. The method described is based on specification patterns supporting an incremental construction of commonly used consistency criteria. We show that our approach fills the gap between a temporal logic such as CTL as a powerful tool for specifying consistency criteria for web documents and users that maintain documents but have no or very limited knowledge about the specification formalism. An empirical assessment of the usability of specification patterns for web documents confirms that a pattern-based specification shows significantly better results than the direct specification with CTL.

1 Introduction

The concept of consistency is commonly applied to databases, programs, protocols, concurrent processes, and systems but can be naturally extended to digital documents. Various notions of consistency and a wide range of consistency checking methods have been studied in the field of digital documents.

In this paper we address the problem of specifying consistency criteria for the purpose of verification of web documents. This work is part of the Verdikt project [19]. We focus on temporal properties of documents along standard reading paths. For example, we check whether in a web-based training (WBT) document every description of a certain concept is followed by an example of the same concept. This kind of consistency is particularly useful when having to ensure document coherence and certain properties of the narrative flow, e.g., in e-learning or technical documentation.

In the Verdikt project the verification is performed by model-checking based on the temporal description logic ALCCTL [18]. For the sake of simplicity and clarity, we express consistency criteria in the more common, but also less expressive computation tree logic - CTL [6, 9] in this paper. However, the results described apply also to ALCCTL. Temporal logics are usually used for verification tasks in the application field of software engineering, but there are also systems using temporal logic for hypertext verification, e.g. [17]. Applying a temporal logic such as CTL or ALCCTL requires good mathematical knowledge and a lot of experience and usually involves considerable effort in terms of manpower and time. For this reason, a high-level mechanism supporting the process of formal specification is highly desirable. Our goal is to provide a user-friendly high-level specification scheme for temporal properties, which supports the incremental construction of commonly used consistency criteria for web documents.

Among the existing methods for high-level specification, pattern-based approaches are well established and widely used [5, 7, 14]. In many cases they do not require deep-level knowledge of the underlying specification formalism. We will show that specification patterns which originally have been introduced for the field of reactive systems [5] can be adapted and enhanced for the purpose of specifying consistency properties of documents. Furthermore, we define an appropriate mapping of patterns onto CTL formulae. We also show that the construction of commonly used consistency conditions for web documents can be performed incrementally, thus giving less experienced users the opportunity to proceed

*This work is partially funded by the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG) under grant number FR 1021/7-2
from low to higher complexity. The usability of specification patterns has been evaluated in comparison to a direct specification using plain CTL. The results show that for inexperienced users the specification with patterns is significantly easier than the one with CTL.

The contribution of this paper consists of:

- defining a set of specification patterns representing general temporal constraints that can be applied to express document consistency,
- showing how the proposed specification patterns can be used in the process of formalizing consistency criteria for web documents, and
- evaluating the usability of the approach.

The paper is organized as follows. Section 2 describes the problem addressed. Section 3 introduces specification patterns for documents, section 4 deals with pattern transformation into CTL, while our specification tool is introduced in section 5. Evaluation results are presented in section 6. Section 7 discusses our results and related work. Section 8 concludes the paper.

2 Problem Description

Our aim is to check the consistency of the narrative structure of a document. The narrative structure represents relevant aspects of the content and structure in a document’s model and is defined as follows (see also [19]).

**Definition 1** (Narrative units. Start units. Narrative relation. Narrative path). A document D is structured as a finite set \( NU \neq \emptyset \) of narrative units where each \( u \in NU \) is a cohesive, self-contained part of D. \( SU \subseteq NU \) denotes a non-empty set of start units such that each \( u \in SU \) is a sensible starting point for reading the document. A narrative relation \( NR \) on the set of narrative units is defined by \( NR := \{(u, u') \in NU \times NU \mid \text{it is sensible to proceed to unit } u' \text{ immediately after having read unit } u\} \). Let \( NR \subseteq NU \times NU \) be a narrative relation of a document. Then a (potentially infinite) sequence \((u_0, u_1, ...)\) of narrative units is a narrative path iff \((u_i, u_{i+1}) \in NR\) for each \( i \in \mathbb{N} \).

**Remark 2** (Narrative path). Since web documents are typically not read linearly, narrative paths cannot be assumed to be acyclic. As a consequence, we have to consider narrative paths to be potentially infinite. By allowing infinite paths the document model does not put a limit on the number of times a certain narrative unit can be visited.

**Definition 3** (Narrative structure). A narrative structure is a tuple \( NS = (NU, SU, NR) \) where \( NU \) is a set of narrative units, \( SU \subseteq NU \) is a set of start units, and \( NR \subseteq NU \times NU \) is narrative relation on \( NU \) such that the following holds:

i) \( NR \) is left-total on \( NU \), i.e. for each \( u \in NU \) there is some \( u' \in NU \) such that \((u, u') \in NR\).

ii) Any narrative unit of \( NU \) can be reached on some narrative path in \( NR \): for each \( u \in NU \) there is a start unit \( s \in SU \) and a narrative path \((u_0, u_1, ...) \in NR\) such that \( u_0 = s \) and \( u_i = u \) for some \( i \in \mathbb{N} \).

**Remark 4** (Narrative structure). Demanding \( NR \) to be left-total simplifies the formal verification framework. For the sake of generality, narrative units which do not have any sensible successor unit are modelled as being reflexively related with themselves.

The process of reading a web document (by a human reader) can be modeled as a collection of paths in a state transition system \((S, R, L)\) where the set of states \( S \) represents the narrative units of the document, \( R \subseteq S \times S \) is a narrative relation on \( S \) and \( L \) represents a set of local interpretations, one for each state \( s \in S \).
**Definition 5** (Temporal structure. Temporal verification model). A CTL temporal structure is a state transition system $M = (S, R, L)$, where:

- $S$ is a set of states,
- $AP$ denotes the set of all atomic CTL propositions,
- $R \subseteq S \times S$ is a left-total binary transition relation, defining the possible transitions between states,
- $L : S \mapsto \mathcal{P}(AP)$ is a labeling function, that assigns each state a set $I \subseteq AP$ of atomic CTL propositions that hold at this particular state.

A **temporal verification model of a document** is a temporal structure with a distinguished starting state $s_0 \in S$.

**Example 6** (Narrative structure). Figure[1] depicts a fragment of a narrative structure of a web document taken from a web based training (WBT) about datastructures. The unit “start” is followed by the “definition of datastructures”. After this unit there are two possible branches to follow. The first one proceeds with an “example of datastructures”, then with a “summary” and a “test about datastructures”, and finally with the “end” unit. The other branch continues with a “definition of abstract datatypes”, followed by an “example of abstract datatypes”, and then joins the other branch at the “summary” unit. Note, that the fact, that a human reader would probably classify the “detour” over the “abstract datatype” unit as a side note, is inessential in our context.

![Figure 1: a) Narrative structure of a document, b) Temporal structure of a document](image-url)

Let us consider the following sample consistency criteria:

1. **On all paths there exists a ”summary” unit before the ”test” unit.**
2. Every ”definition of the topic datastructure” is on all succeeding paths followed by an ”example” of the same topic.
3. After the ”summary” unit, no ”definition” units are allowed.
Obviously, criterion 1 holds in the structure of Figure 1a). On both paths a "summary" unit exists immediately before the "test" unit.

On the other hand, criterion 2 does not hold in the given structure, because there is a path ("start", "definition of datastructure", "definition of abstract datatypes", "example of abstract datatypes", "summary of datastructure", "test about datastructure", "end") with a "definition of datastructure" not being followed by an "example of datastructure".

Finally, criterion 3 holds in the given structure, because both definitions ("definition of datastructure", "definition of abstract datatypes") appear before the "summary" unit, concerning both possible paths.

**Example 7** (Temporal structure of a document). Figure 1b) depicts the CTL temporal structure of a document with a narrative structure as shown in Figure 1a). In this structure holds:

- the set of states is defined as \( S = \{ s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7 \} \),
- atomic propositions correspond to topics (datastructure, abstract datatype) and structural types (definition, example, summary, ...),
- the transition relation is given by \( R = \{ (s_0, s_1), (s_1, s_2), (s_1, s_3), ..., (s_6, s_7), (s_7, s_7) \} \),
- the labeling function determining the interpretation of a state is defined as \( L = \{ s_0 \mapsto \{ \text{start} \}, s_1 \mapsto \{ \text{definition, datastructure} \}, ..., s_7 \mapsto \{ \text{end} \} \} \).

There are two different paths, namely \( s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow s_5 \rightarrow s_7 \rightarrow ... \) (short for: \( \{(s_0, s_1), (s_1, s_2), (s_2, s_3), (s_3, s_4), (s_4, s_5), (s_5, s_7), (s_7, s_7)\} \)) and \( s_0 \rightarrow s_1 \rightarrow s_3 \rightarrow s_4 \rightarrow s_5 \rightarrow s_6 \rightarrow s_7 \rightarrow ... \).

The main steps of consistency specification and verification are shown in Figure 2. Users appear in two different roles: First, there are document authors, who provide, organize, and maintain document fragments. Experienced authors may also be able to specify consistency criteria using the interface for pattern-based specification to be described later in this paper. Second, there are temporal logic experts who can specify complex criteria directly in CTL and maintain the verification model, if necessary.

![Figure 2: Automated verification of semi-structured documents](image)

Assume that there are several text components, possibly in different formats (no. 1 in Figure 2). The information about the document’s content and structure are available in the form of markup and external metadata or are provided by external information extraction tools. The collected information is represented by a *temporal verification model* (no. 2 in Figure 2) which essentially formalizes the
narrative structure of the document. This way an abstraction is provided from implementation details which are irrelevant for the verification tasks.

The specification criteria are expressed in CTL (no. 3 in Figure[2]) and verified against the verification model by the CTL model checker. The verification results (counterexamples) are then presented to the user (no. 4 in Figure[2]). For example, a CTL formula which expresses the second criterion of Example[6] reads: $\text{AG}((\text{definition} \land \text{datastructure}) \rightarrow \text{AF}((\text{example} \land \text{datastructure}))$)

Since CTL, as a temporal logic, is likely to be too demanding for non-expert users - which of course tend to be the majority - a user-level specification method based on specification patterns has been developed (no. 5 in Figure[2]). Patterns represent commonly occurring requirements concerning the content and structure of documents (see Definition[9]). Specification patterns are translated into CTL formulae. Our approach to automated verification of semi-structured documents is presented in detail in [19].

3 Specification Patterns for Documents

The primary goal of the work described in this paper is the definition of a high-level specification formalism for consistency criteria for web documents, which should fulfill the following properties:

- The proposed high-level formalism must represent the temporal properties of web documents and must be intuitively understandable, so that a user does not have to be aware of the underlying logic formalism.

- The system should provide a reasonable expressive power and yet stay compact and manageable.

- It is important to support the incremental development of specifications, i.e., it should allow the user to first recognize the general rule and then to refine it if required.

- The approach has to be extensible and adaptable to possibly different underlying logic formalisms.

A pattern-based approach to the presentation, formulation, and reuse of property specifications in reactive systems has been introduced in [5]. A set of possible constraints has been defined and patterns have been created for them. The patterns are provided to the users who can identify similar requirements in their systems and select patterns that address those requirements. Until now, seven specification formalisms are supported, among them CTL [11]. We found that many of these patterns could also be useful for expressing document properties [10] [13]. The abstraction from temporal properties allows users not to worry about the underlying logic. The flexible definition and organization of the original patterns allow us to choose only a subset and to adapt and extend it easily for our needs.

Because patterns defined in [5] are meant to be used by users familiar with the underlying specification formalism, user support for the specification process is not provided. Different from that situation, our use cases (see [15]) involve non-expert users; consequently, we have to support them in expressing formal consistency criteria. To this end, we provide an interface allowing to express loose criteria, which can be later enhanced if necessary.

Example 8 (Properties of consistency criteria). Let us consider the consistency criterion: There always exists a "summary" unit before the first "test" unit. The following important properties can be observed:

1. It expresses a kind of constraint: the existence of a "summary" unit.

2. It specifies the part of the document or, more precisely, of its temporal structure, where the specification should hold: before the first "test" unit.

The properties 1. and 2. characterize a specification pattern of the following kind: Within the considered structure, on all paths starting from the current state, property p holds before property s holds for the first time. The considered structure can be the whole document, but also any document fragment.
As one can observe, criteria expressed in natural language are quite ambiguous. For example, requiring that each "definition of datastructure" is followed by an "example" on the same topic does not specify precisely whether there should be an "example of datastructure" on all following paths after the "definition of datastructure", or whether it is enough having an example on some path.

Natural language specifications of certain properties of specification patterns are also ambiguous. Here are some examples of such ambiguities:

- Does $q$ follows $p$ require that $q$ has to hold on all following paths, or on some path?
- After $s$ could mean after each $s$ or after the first one. It is also not clear what happens if there is no $s$ in the whole document. Is the criterion satisfied in this case or not?
- Does the meaning of before $s$ include the narrative unit where $s$ holds for the first time or not?

The ambiguities of natural language specifications were the main motivation for us to first define a set of basic specification patterns together with their corresponding CTL formulae and then to determine how the basic patterns can be modified, i.e. we defined a set of modified patterns with their corresponding CTL formulae. This way users can execute a two-stage process, first determining the general properties of the criterion they want to express adding refinements as necessary in the second step.

The semantics of pattern types, scopes, and modifiers we use is determined by the definition of the mapping of specification patterns onto CTL as will be detailed in section 4.

**Definition 9** (Specification pattern). A specification pattern (for documents) is a generalized representation of a commonly occurring requirement on the content and structure of documents (cf. [5]). Specifications are instances of specification patterns. A specification pattern is represented by a 4-tuple: $(\text{pattern type}, \text{p modifier}, \text{scope}, \text{s modifier})$.

- A pattern type $(\text{pattern type})$ determines the type of the constraint expressed by the specification pattern. Each pattern type is represented by a pattern type name and one or two pattern properties. Pattern type names (universally, exists, follows, precedes) denote the type of the constraint and can only be understood in conjunction with pattern properties. A pattern property is a parameter which represents the CTL formula required to hold by the pattern type. Let $p$ and $q$ be CTL formulae. Possible values of pattern type are: universally $p$, exists $p$, $q$ follows $p$, and $p$ precedes $q$.
  
  - universally $p$ means that $p$ holds in every narrative unit.
  - exists $p$ expresses that $p$ has to hold in some narrative unit.
  - $q$ follows $p$ means each unit satisfying $p$ must be succeeded by a unit for which property $q$ holds.
  - $p$ precedes $q$ means that if property $q$ holds in some narrative unit this unit must be preceded by a unit for which property $p$ holds. By default, each pattern type applies to all paths of a document but this can be overridden.

<table>
<thead>
<tr>
<th>pattern type</th>
<th>pattern modifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>universally $p$</td>
<td>null, absence, some path</td>
</tr>
<tr>
<td>exists $p$</td>
<td>null, some path</td>
</tr>
<tr>
<td>$q$ follows $p$</td>
<td>null, immediate</td>
</tr>
<tr>
<td>$p$ precedes $q$</td>
<td>null</td>
</tr>
</tbody>
</table>

Table 1: Pattern types with allowed pattern modifiers

- A pattern modifier $(p$ modifier) allows to refine a pattern type, by further restricting or loosening the original meaning. Possible values of $p$ modifier are: null, absence, immediate, some path. Modifier null indicates that the original meaning of a pattern type is not changed.
Table 1 shows the allowed pattern modifiers for each pattern type. For pattern types universally $p$ and exists $p$ there is a pattern modifier some path. It says that the constraint holds on some path of a document, as opposed to the default meaning. For the pattern type universally $p$ a pattern modifier absence is defined, which denotes that $p$ does not hold in any narrative unit. The pattern type $q$ follows $p$ can be used with the modifier immediate, which expresses that $q$ must hold in all next narrative units of the one where $p$ holds.

- A scope determines where in a document a specification is intended to hold. A scope is represented by a scope name and one or two scope properties. A scope property is a parameter, which will be replaced by a CTL formula at instantiation time. Let $s$ and $r$ be CTL formulae. Possible values of scope are: globally, before $s$, after $s$, and between $s$ and $r$.

Scope globally requires no parameters and actually expresses an unrestricted scope - a specification having this scope applies to the whole document structure. before $s$ expresses that the specification holds before or in the same narrative unit where $s$ holds for the first time. Similarly, after $s$ requires that the specification holds after or in the same narrative unit where $s$ holds for the first time. Scope between $s$ and $r$ denotes each part of a document structure between an appearance of property $s$ and the first following appearance of property $r$.

Table 2 shows allowed combinations of pattern types and scopes. Every pattern type can be combined with scopes globally, before $s$, and after $s$. Pattern types universally $p$ and exists $p$ can also be used with scope between $s$ and $r$.

<table>
<thead>
<tr>
<th>pattern type</th>
<th>scopes</th>
</tr>
</thead>
<tbody>
<tr>
<td>universally $p$</td>
<td>globally, before $s$, after $s$, between $s$ and $r$</td>
</tr>
<tr>
<td>exists $p$</td>
<td>globally, before $s$, after $s$, between $s$ and $r$</td>
</tr>
<tr>
<td>$q$ follows $p$</td>
<td>globally, before $s$, after $s$</td>
</tr>
<tr>
<td>$p$ precedes $q$</td>
<td>globally, before $s$, after $s$</td>
</tr>
</tbody>
</table>

Table 2: Pattern types with allowed scopes

- A scope modifier ($s$ modifier) allows the refinement of a scope by further restricting the original meaning. Possible values of $s$ modifier are: null, real before, and real after. Modifier null indicates that the original meaning of a scope is not changed.

Table 3 shows the allowed scope modifiers for each scope. Scope before $s$ can be restricted with a scope modifier real before to express that the constraint expressed by the pattern type holds really before $s$, i.e. no later than in the preceding unit of the one at which $s$ holds. Similarly, scope after $s$ can be restricted with a scope modifier real after to express that it is not sufficient that the constraint represented by the pattern type holds in the same unit with $s$, but only after it.

<table>
<thead>
<tr>
<th>scope</th>
<th>scope modifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>globally</td>
<td>null,</td>
</tr>
<tr>
<td>before $s$</td>
<td>null, real before</td>
</tr>
<tr>
<td>after $s$</td>
<td>null, real after</td>
</tr>
<tr>
<td>between $s$ and $r$</td>
<td>null,</td>
</tr>
</tbody>
</table>

Table 3: Scopes with allowed scope modifiers

23
Specification patterns of the form \((\text{pattern}\_\text{type}, \text{null}_p, \text{scope}, \text{null}_s)\), where both modifiers are set to \text{null}, are called \textit{basic specification patterns}, while the others are \textit{modified specification patterns}.

According to Tables 1, 2, and 3 there are 45 specification patterns for documents, 14 of which are basic specification patterns.

\[ s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow s_7 \rightarrow s_8 \rightarrow s_9 \rightarrow ... \]

\[ s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow s_5 \rightarrow s_6 \rightarrow s_7 \rightarrow s_8 \rightarrow s_9 \rightarrow ... \]

\[ s_0 \rightarrow s_7 \rightarrow s_8 \rightarrow s_9 \rightarrow ... \]

\[ \text{c1} \] There is always a "test" unit before the first "definition" unit.
This criterion requires that a "test" exists before the first "definition". Obviously, the specification pattern of type \( \exists \) and scope before \( s \) is needed: \( (\exists \text{ test}, \text{null}_p, \text{before definition}, \text{null}_r) \). The corresponding CTL formula\(^1\) reads: \( A[\neg \text{definition} W \text{ test}] \)

**c2 Every "definition of the topic datastructure" is followed by an "example of a datastructure".**

It is required that every "definition of datastructure" is followed by an "example of datastructure". This corresponds to the pattern type \( q \) follows \( p \) and scope globally (this requirement concerns the whole document): \( ((\text{example} \land \text{datastructure}) \text{ follows } (\text{definition} \land \text{datastructure}), \text{null}_p, \text{globally}, \text{null}_r) \). The corresponding CTL formula reads:

\[
AG((\text{definition} \land \text{datastructure}) \rightarrow AF(\text{example} \land \text{datastructure}))
\]

**c3 Each unit between the "start" unit and "summary of datastructure" is dealing with datastructures.**

This criterion corresponds to the pattern type universally \( p \) and scope between \( s \) and \( r \). "Datastructure" must hold within each narrative unit between the "start" unit and "summary of datastructure": \( (\text{universally datastructure}, \text{null}_p, \text{between start and summary datastructure}), \text{null}_r) \). Note that due to the pattern modifier \( \text{null}_p \) this pattern indeed requires the pattern formula to hold on all paths. The corresponding CTL formula reads:

\[
AG((\text{start} \land \neg(\text{summary} \land \text{datastructure})) \rightarrow A[\text{datastructure} W (\text{summary} \land \text{datastructure})])
\]

Criterion **c1** holds in the temporal structure in Figure [3](#). On paths \( p_1 \) and \( p_2 \) the first "definition" is found in unit \( s_1 \) and there is a "test" before it (unit \( s_1 \)). On path \( p_3 \) there is no "definition" and thus the criterion holds by convention.

Also criterion **c2** holds in the temporal structure of Figure [3](#). There is one "definition of datastructure" (unit \( s_1 \)), which is followed by an "example" on the same topic (unit \( s_2 \)) on the relevant paths \( p_1 \) and \( p_2 \). Note that there is also an "example of datastructure" before the "definition", which does not affect the validity of the criterion.

Criterion **c3** does not hold in the temporal structure in Figure [3](#). On path \( p_2 \) there are two narrative units (\( s_5 \) and \( s_6 \)) between "start" and "summary of datastructure" at which datastructure does not hold.

In the sequel we present some examples of modified specification patterns. To better explain the difference in the meaning between basic and modified specification patterns we also show the corresponding CTL formulae.

**Example 11** (Modifier immediate\(_p\)). Consider the following constraints:

1. Every "definition of the topic datastructure" has to be followed on all paths by an "example" on the same topic. To express this constraint we use the pattern - \( ((\text{example} \land \text{datastructure}) \text{ follows } (\text{definition} \land \text{datastructure}), \text{null}_p, \text{globally}, \text{null}_r) \). The corresponding CTL formula reads: \( AG((\text{definition} \land \text{datastructure}) \rightarrow AF(\text{example} \land \text{datastructure})) \)

The temporal operator \( \text{F} \) expresses that an example of a datastructure holds eventually in some narrative unit.

2. Every "definition of the topic datastructure" has to be immediately followed (i.e. in each next narrative unit) by "examples" on the same topic. The pattern used above has to be modified with immediate\(_p\): \( ((\text{example} \land \text{datastructure}) \text{ follows } (\text{definition} \land \text{datastructure}), \text{immediate}_p, \text{globally}, \text{null}_r) \). In the previous CTL formula the temporal operator \( \text{F} \) (eventually) is replaced by \( X \) (next): \( AG((\text{definition} \land \text{datastructure}) \rightarrow AX(\text{example} \land \text{datastructure})) \)

Both constraints hold in the temporal structure of Figure [3](#)

\(^1\)W - weak until: \( A[p W q] := \neg E[\neg q U (\neg q \land \neg p)] \)
Example 12 (Modifier real_before). Consider the criterion: *there is always a "summary" unit before the first "test"*. To represent it, we can use the specification pattern: \((\text{exists } \text{summary}, \text{null}_p, \text{before } \text{test}, \text{null}_s)\). The corresponding CTL formula reads: \(A[\neg \text{test} W \text{summary}]\)

The meaning of the scope \(\text{before } s\) implies that "test" and "summary" could actually hold in the same narrative unit. To express the more strict specification, that "summary" occurs really before "test" (no later than in the preceding unit) we use the specification pattern: \((\text{exists } \text{summary}, \text{null}_p, \text{before } \text{test}, \text{real}_\text{before})\). The corresponding CTL formula reads: \(A[\neg \text{test} W (\text{summary} \land \neg \text{test})]\)

4 Pattern Transformation to CTL Formulae

The meaning of a specification pattern is determined by its mapping onto a CTL formula. The mappings of specification patterns onto a CTL formulae are stored in the table of mappings. For every pattern, there is exactly one formula. Due to space constraints, Table 4 shows only a part of table of mappings with four patterns described in section 3. Columns one through four represent the specification pattern (pattern type, pattern modifier, scope, and scope modifier, respectively), and column five contains the corresponding CTL formula. The complete table of mappings can be found in [12].

<table>
<thead>
<tr>
<th>pattern type</th>
<th>pattern modifier</th>
<th>scope</th>
<th>scope modifier</th>
<th>CTL formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>exists (p)</td>
<td>\text{null}_p</td>
<td>before (s)</td>
<td>\text{null}_s</td>
<td>(A[\neg s W p])</td>
</tr>
<tr>
<td>exists (p)</td>
<td>\text{null}_p</td>
<td>before (s)</td>
<td>real_before</td>
<td>(A[\neg s W (p \land \neg s)])</td>
</tr>
<tr>
<td>(q) follows (p)</td>
<td>\text{null}_p</td>
<td>globally</td>
<td>\text{null}_s</td>
<td>(AG(p \rightarrow AFq))</td>
</tr>
<tr>
<td>(q) follows (p)</td>
<td>immediate_p</td>
<td>globally</td>
<td>\text{null}_s</td>
<td>(AG(p \rightarrow AXq))</td>
</tr>
</tbody>
</table>

Table 4: Table of mappings (partial)

Every specification pattern is mapped onto exactly one CTL formula but not all CTL formulae can be represented in the form of a pattern instance. For example, there is no corresponding specification pattern for the following CTL formula: \(AG EF \text{help}\) (At any point help is eventually reachable). This problem could be solved by introducing a new specification pattern, or by allowing the composition of existing patterns. However, there is a tradeoff between expressiveness and usability of the pattern system which we dealt with in favor of usability.

5 Specification Tool

A first prototype of a specification tool provides basic support for the specification process and helps users to incrementally build a specification pattern.

Figure 4 shows a screen-shot of the GUI. Before building the specification, the user chooses the document to be verified (no. 1 in Figure 4). After that, the process of constructing a specification starts. First, the user chooses the constraint type she wants to express (i.e. pattern type) - component 2 in Figure 4. For each pattern type, there is an explanation of its meaning. Second, a pattern modifier is to be set - component 3 in Figure 4. Only allowed modifiers for the previously chosen pattern type are enabled. The appropriate scope is to be chosen as the third component (no. 4 in Figure 4). The last component of the specification pattern is a scope modifier (no. 5 in Figure 4). Again, only the allowed scope modifiers for the already chosen scope are enabled.

After having chosen the complete specification pattern, the user is presented with the natural language formulation of this pattern with placeholders (no. 6), which are to be bound to atomic propositions from
the temporal model. For the inspection of the temporal model, a dedicated additional tool is provided [16]. Finally, the CTL formula corresponding to the constructed and refined specification is shown (no. 7). Having finished the specification, the user activates the model checker (no. 8).

**Example 13** (Construction of a consistency criterion). Let us assume that the user wants to specify the following constraint: *On all paths there exists a "summary" unit before the first "test" unit. "Summary" unit and "test" unit may not occur in the same narrative unit.* The following steps are to be performed:

1. Choose the document to be verified (no. 1 in Figure 4).
2. Choose the pattern type \( \exists p \) (no. 2 in Figure 4). This pattern type has one corresponding parameter (P) which will be instantiated in step 5.
3. Choose the pattern modifier \( \text{null}_p \) (no. 3 in Figure 4).
4. Choose the scope \( \text{before } s \) (no. 4 in Figure 4). This scope has one corresponding parameter (S) which will be instantiated in step 6.
5. Choose the scope modifier \( \text{real}_{\text{before}} \) (no. 5 in Figure 4).
6. The corresponding natural language phrase reads (no. 6 in Figure 4):

   \[
   \text{On all paths, } P \text{ holds eventually, before } S \text{ holds for the first time.}
   \]

   In our example, the user replaces \( P \) by the atomic proposition \( \text{summary} \) and \( S \) by the atomic proposition \( \text{test} \).
7. Look up the respective CTL formula from the translation table (cf. Table 4) and replace variables with atomic propositions determined in step 6: $A[\neg \text{test} W (\text{summary} \land \neg \text{test})]$

8. Verify if the specified criterion holds in the chosen document.

The steps 1 to 5 are performed by the user. In step 6 both the system and user participate, while the system performs steps 7 and 8.

Our specification tool was implemented in Java 1.6. For model checking we used the CTL model checker NuSMV [4].

6 Evaluation

We evaluated the method of pattern-based specification of consistency criteria as compared to a direct specification using plain temporal logic CTL. Goals of the evaluation were:

- to show that even inexperienced users, after receiving instructions about specification patterns and their usage, can successfully use them,
- to show that under comparable conditions, the application of specification patterns by inexperienced users leads to remarkably better results than the usage of plain temporal logic like CTL.

The evaluation has been conducted with 108 volunteer participants, all of which were students from various fields of study. The participants were split into two groups of 54 members each. No participants had previous experience with either CTL or specification patterns for web documents.

Test questions addressed the formulation of consistency criteria concerning a single test document. Both groups had to specify the same five consistency criteria. The first group (control group) was asked to specify criteria using CTL whereas the second group (experimental group) had to apply specification patterns. The test document was a user manual for a digital camera, found on the manufacturer web site; for details see [11].

The test was performed separately for each group. At the beginning, both groups were given an introduction to the Verdikt project and the test environment. Afterwards, the first group attended a practical compact training course in CTL (ca. 45 minutes). The CTL-syntax and semantics were explained on a rather intuitive level relying mostly on examples and graphical illustrations. Some examples shown had a structure similar to the test questions. After the CTL training, the participants of the first group were asked to answer the test questions separately and individually, i.e., without team work. An overview of CTL operators was available to each student for reference.

The second group was introduced to the structure and meaning of specification patterns as well as examples of their usage (ca. 30 minutes). As for the first group, some examples shown had a structure similar to the test questions. As reference material a list of all specification patterns was available.

We validated the answers as either usable or unusable. A specification was considered usable, if no "false positives" resulted when using it in a verification run. However, we classified as acceptable "false negatives" that were produced when applying a specification that was usable in the aforementioned sense. That is, specifications that were stricter than required were classified as usable and weaker specifications were classified as unusable.

The validation results confirm the usability of our approach. The results show that after having received brief instructions about specification patterns even inexperienced users can use them with a success rate of over 70%. The validation results confirm also that for inexperienced users it is considerably easier to express the criteria by using specification patterns as compared to using CTL formulae. Under comparable conditions, participants of the first group (CTL) could only answer ca. 32% of all
questions correctly, while for the second group the success rate was over 70%. Precise validation results are presented in Table 5 and in the diagram of Figure 5.

<table>
<thead>
<tr>
<th># correctly answered questions per participant</th>
<th># participants from group 1 (CTL)</th>
<th>group 2 (patterns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>totally participants</td>
<td>54</td>
<td>54</td>
</tr>
</tbody>
</table>

Table 5: Validation results

Figure 5: Number of correctly specified criteria per participant

The validation of test results has also confirmed our assumption that the instantiation of parameters should be additionally supported. This problem is addressed by the Verdikt project [16].

7 Discussion

The described temporal patterns with CTL as the underlying formalism are adequate for expressing path-oriented temporal criteria for web documents. Criteria also containing semantical dependencies, like, e.g., every definition of some topic is on all succeeding paths followed by an example of the same topic, cannot be expressed in CTL. Within the Verdikt project, we use a temporal description logic ALCCTL [18] as a specification formalism. Our patterns can also be used with ALCCTL [19], but in this case the instantiation of parameters becomes rather complex and therefore has to be additionally supported. To this end, in our ongoing research we are extending the method described here towards a user-friendly representation of ontological knowledge.

If a criterion does not hold in the temporal structure, the model checking results in a rather technical counter-example, which is not very informative for a user. For this reason, an incremental construction of counterexamples has been defined [20].

The prototype of the specification tool presented in section 5 shows only one rather simplified way
to support the usage of specification patterns. Our ongoing research considers also an example-based specification method.

Our approach to document verification by model checking is not targeted at the XML data model of ordered trees but at documents with a graph (but not necessarily tree) structure such as hypertext. Properties of paths in such graph-structured documents are hard to express and inefficient to check using XPath and first order logic. A detailed study of the expressiveness and performance of our approach as compared to methods based on XML processing is presented in [18].

The problem of high-level specification appears in different research areas and there are also diverse approaches to its solution. Diagram-based languages have been suggested in the areas of real time systems [2] and workflow modeling [3]. These graphical languages are closely related to the underlying formalism and, as a result, it is impossible to interpret or create diagrams without deep knowledge about the logic they represent.

Dwyer et al. were the first to present specification patterns for temporal properties [5]. Many other researchers dealing with temporal specifications have adapted these patterns to different purposes. In the sequel we refer to two of them. In [7] the authors adopted the idea of specification patterns. An interactive visual framework that employs structured English sentences as a user front-end for the specification of Clocked CTL (CCTL) formulae for model-checking has been developed. [14] deals with timing-based requirements for embedded systems. The authors present real-time specification patterns in terms of three commonly used real-time temporal logics (MTL, TCTL, RTGIL). In addition, they have developed a structured English grammar, to further facilitate the understanding of a specification meaning. In the application field of web documents, which could possibly have a very complex structure with many branches, the representation of patterns only by structured language is not sufficient. Our ongoing research considers an additional example-based support for the specification process.

Conceptual authoring [8] is a method which uses a natural language text as a means of presenting semantic content during knowledge editing. By this method all editing operations are defined directly on an underlying logical representation, governed by a predefined ontology. By using common generic phrases users need not be aware of the underlying formalism. We will evaluate conceptual authoring within our ongoing research concerning parameter instantiation, as mentioned above.

8 Conclusion and Outlook

A user-friendly method for the high-level specification of consistency criteria for web documents has been presented and its usability has been shown. We define specification patterns for web documents as a high-level formalism for consistency criteria. Patterns hide the underlying logic formalism and are represented by simple natural language expressions, like, e.g., *existence before*. They also allow for the incremental building of specifications. This is especially convenient for users not familiar with temporal logics and can make all the difference between using temporal logic for consistency checking and ignoring it altogether. The usability of our approach has also been demonstrated. We believe that we found a good balance between expressive power and usability. If necessary, the system can be extended with new patterns and also adapted to another underlying formalism.

In future work we will adapt our patterns to the temporal description logic ALCCTL [18] to increase their expressive power. First experiments let us expect that the proposed specification patterns and specification environment help users to formalize application-specific constraints on documents. We will also examine the possibility of composing of specification patterns. The usability of patterns is going to be increased by an example-based specification method.
9 Acknowledgments

Dr. Petia Genkova, University of Passau, contributed to the evaluation with valuable insights.

References


Specifying Hyperdocuments with Algebraic Methods

Volker Mattick
Technische Universität Dortmund
Fakultät für Informatik
volker.mattick@tu-dortmund.de

Abstract

Algebraic specification methods, well-known in the area of programming languages, are adapted to present a tailored framework for hyperdocuments and hyperdocument systems. In this framework, a hyperdocument is defined via its abstract syntax, which is a variable-free term of a suitable constructor-based signature. Both the representation in a markup language and the graphical presentation on the screen as well as further representations are elements of particular algebraic interpretations of the same signature. This technique allows the application of well-known methods from the field of compiler construction to the development of hyperdocument systems. Ideas for its implementation in the functional language Haskell are roughly drafted. It is shown how XML-based markup languages with schemas and stylesheets can be defined in terms of this framework and how this framework can be extended so that it can deal with partially specified documents, called semi documents. These semi documents can be automatically adapted to the users’ needs, which e.g. is helpful to ensure accessibility.

1 Introduction

A hyperdocument\footnote{The term hyperdocument is used here as a generalization of the terms hypertext document and hypermedia document.} is a particular electronic document, which one does not have to read in a linear way. Parts of the document, referenced by anchors, are connected with other documents or parts of them via hyperlinks, which induce a hyper structure on the text. Hyperdocument engineering is a special discipline of software engineering\cite{17}. Instead of general programming languages and assembly languages, markup and layout languages are used; instead of context-free grammars, there are schemas; instead of compilers, browsers transform source documents from a markup language into a layout language. The life cycle for hyperdocuments is much shorter than for most other kinds of software products\cite{14}, and in no other field of software engineering have programmers, designers and users become more closely related over the last decade than in hyperdocument engineering. The differences between producers and consumers blur so much that the term prosumer\cite{13} is sometimes used.

Software engineering in general\cite{27} and compiler construction in particular\cite{25,26} benefit from the rigorous use of algebraic specification techniques, with e.g. concepts like constructor-based signatures, syntax trees, morphisms, semantic interpretations, specifications and refinement\cite{5}, which originate mainly in the area of mathematical logic and universal algebra\cite{10}. A clean separation between the level of modeling and the level of implementation is always recommended. With a precise and clear model, implementation becomes much easier and more reliable.

This work adapts general algebraic methods to define a tailored framework for hyperdocuments and hyperdocument systems and shows how hyperdocument engineering can benefit from this approach.

2 Preliminaries

Research and industry have created many different approaches for modeling hyperdocuments, starting with Vannevar Bush’s concepts in 1945, continuing with the Dexter model and the tower model in the nineties of the last century, and leading to the document object model and area model of the WWW era.
The models can be categorized into two classes, the information-centered models, which focus on the structure of a hyperdocument, and the screen-based models, which focus on its presentation.

From a technical point of view, hyperdocuments are a particular kind of electronic documents. With the growing success of the WWW, markup languages have practically become the standard for the concrete description of the information-centered aspect of hyperdocuments. The screen-based aspects are usually described by layout languages. Both are subclasses of the context-free languages. A hyperdocument system is a tool that transforms a hyperdocument from one representation into another. Today, the most commonly used hyperdocument systems are browsers. They take a hyperdocument, coded in a markup language, and transform it into a visual presentation of this document.

![Figure 1: Hyperdocuments](image)

An (algebraic) specification $SP = (\Sigma, \mathcal{A})$ consists of a (many-sorted) signature $\Sigma$, which captures the abstract syntax, and a set of $\Sigma$-algebras $\mathcal{A}$ which capture the allowed semantics. The $\Sigma$-algebras can either be given explicitly by writing them as tuples of carrier sets and functions, or implicitly by a set of $\Sigma$-Horn formulas.

A (many-sorted) signature $\Sigma = (S, F, R)$ consists of a set $S$ of sorts, an $S^* \times S$-sorted set $F$ of function symbols, and an $s$-sorted set $R$ of relation symbols. If $R = \emptyset$, we write $\Sigma = (S, F)$ for short. Instead of $f \in F_{(s,s)}$, we write $f : e \to s \in F$. If $e = e$, then $f$ is called a constant. Function symbols that only build up data are called constructors and are denoted by $CO$, other function symbols are called defined functions and are denoted by $DF$. If $F = CO$, that means all function symbols are constructors, then the signature is called a constructor signature; if $F = CO \cup DF$, it is called a constructor-based signature.

If $X$ is an $s$-sorted set of variables, the $s$-sorted set $T_\Sigma(X)$ of $\Sigma$ terms is defined inductively, so that for all $s \in S$ holds $X_s \subseteq T_\Sigma(X)_s$, and for all $w \in S^*$, $s \in S$, $f : w \to s \in \Sigma$ and $t \in T_\Sigma(X)_w$ holds $f(t) \in T_\Sigma(X)_s$. The set of variable-free terms, called $\Sigma$ ground terms, is denoted by $T_\Sigma$. The $s$-sorted set $At_\Sigma(X)$ of $\Sigma$ atoms is defined inductively, so that for all $w \in S^*$, $r : w \in \Sigma$ and $t \in T_\Sigma(X)_w$ holds $r(t) \in At_\Sigma(X)$. The set of variable-free atoms, called $\Sigma$-ground atoms, is denoted by $At_\Sigma$.

A $\Sigma$-algebra is a tuple $\mathcal{A} = (A, OP)$, where for all $s \in N$ there exists exactly one non-empty carrier set $A_s \in A$, for all $(co : s) \in CO$ there exists exactly one element $co^A \in A_s$, and for all $(co : e \to s) \in CO$ there exists exactly one function $co^A : A_e \to A_s \in OP$. A function $\sigma$, which assigns an algebra to a given signature, is called a semantic function. A $\Sigma$-algebra $\mathcal{A}_{init}$ is said to be initial in the class of all $\Sigma$-algebras if there is for each $\Sigma$-algebra $\mathcal{A}$ exactly one $\Sigma$-homomorphism $init : \mathcal{A}_{init} \to \mathcal{A}$, called initial homomorphism. A $\Sigma$-homomorphism is an $s$-sorted mapping $h : \mathcal{A}_1 \to \mathcal{A}_2$ between two $\Sigma$-algebras $\mathcal{A}_1$ and $\mathcal{A}_2$, so that for all function symbols $f : w \to s \in \Sigma$ holds $h_1 \circ f^{A_1} = f^{A_2} \circ h_w$ \[10\]. The initial algebra, if it exists, is unique up to isomorphism. So in Fig. 2 $\sigma_{init}$ is the semantic function, which assigns the initial $\Sigma$-algebra $\mathcal{A}_{init}$ to the signature $\Sigma$, and each $h_i$ is an initial morphism. Because $\sigma_{init}$ is unique if it exists, and, according to the definition, there is exactly one $h_i$, it holds $\sigma_i = \sigma_{init} \circ h_i$, and so it is possible to define the semantics of a signature via the initial algebra. It can be proven that for each constructor signature, the set of all ground terms is initial in the class of all $\Sigma$-algebras \[10\].

---

*Footnote: If $\mathcal{A}$ is a singleton, the name of the set of algebras and the name of the algebra are identical.*
\(\Sigma\)-Horn formulas [24] are of the form \(f(t_1, \ldots, t_n) \equiv u \iff G\) or \(r(t_1, \ldots, t_n) \iff G\), where \(f \in DF\), \(t_1, \ldots, t_n, u\) are constructor terms, \(G\) is a conjunction of \(\Sigma\) atoms, called goal, and \(\text{var}(u) \subseteq \text{var}(t_1, \ldots, t_n, G)\), where \(\text{var} : T_\Sigma \rightarrow \mathcal{P}(X)\) denotes the set of variables in a constructor term.

A specification can be built hierarchically. Given a specification \(SP' = ((S', F'), (A', OP'))\), a set \(S\) of sort symbols disjoint from \(S'\), and a set \((S' \cup S^\ast) \times (S' \cup S^\ast)\)-sorted function symbols \(F\) disjoint from \(F'\), the specification \(SP_{SP'} = (\Sigma_{SP'}, \varphi_{SP'})\) with \(\Sigma_{SP'} = (S' \cup S^\ast, F \cup F')\) and \(\varphi_{SP'} = (A \cup A', OP \cup OP')\) is a \(\Sigma_{SP'}\)-algebra called the specification over basic specification \(SP'\).

A compiler translates source code, written in a concrete syntax of a source language, usually described by a context-free grammar, into a semantically equivalent artifact of a target language. The first part of a compiler is called front-end or parser, denoted by \(\text{parse}_{L(G)}\). It analyses the source code, singles out erroneous words regarding the given language \(L(G)\), and assigns to each valid word \(w \in L(G)\) a tree-structured representation \(t\), called (abstract) syntax tree. The second part is called back-end or evaluator, denoted by \(\text{eval}^{L'}\). It assigns to each syntax tree \(t\), possibly enriched with additional attributes, artifacts of the target language \(L'\). For particular classes of context-free grammars, it can be shown that \(\text{parse}_{L(G)}\) can be automatically generated from the grammar \(G\). In these cases, it suffices to define a compiler by a tuple \((G, \text{eval}^{L'})\).

A context-free grammar is a tuple \(G = (N, T, P, S)\), where \(N\) is a finite set of non-terminal symbols, \(T\) is a finite set of terminal symbols, \(P\) is a finite set of production rules, \(S \in N\) is the start symbol, and all production rules are of the form \(X \longrightarrow w\), where \(X \in N\) and \(w \in (N \cup T)^\ast\). It is called a (linear) tree grammar if all production rules have the form \(X \longrightarrow tw\), where \(X \in N\), \(t \in T\) and \(w \in N^\ast\), and for all production rules \(X \longrightarrow tw\) and \(X' \longrightarrow tw'\), \(X, X' \in N\), \(w, w' \in N^\ast\), \(t \in T\) holds \(X = X'\).

A grammar over a basic specification is a tuple \(G(SP') = (N \cup S', T \cup A', P, S)\), where \(N\) is a finite set of non-terminal symbols disjoint from \(S'\), \(T\) is a finite set of terminal symbols disjoint from \(A'\), \(P\) and \(S\) are as previously defined, and \(SP' = ((S', F'), (A', OP'))\). In a context-free grammar over a basic specification, all rules have the form \(X \longrightarrow \varepsilon\) or \(X \longrightarrow w_1X_1 \ldots w_nX_nw_{n+1}\) with \(X, X_i \in N\) and \(w_i \in (N \cup S' \cup T \cup A')^\ast\) for \(1 \leq i \leq n\). In a tree grammar over a basic specification, all rules have the form \(X \longrightarrow tw\), with \(X \in N\), \(t \in T \cup A'\) and \(w \in (N \cup S')^\ast\) for \(1 \leq i \leq n\).

Context-free grammars are usually described in a metasyntax named Backus-Naur Form (BNF cf. [16]), or in Extended Backus-Naur Form (EBNF cf. [9]). Also, the Augmented Backus-Naur Form
(ABNF cf. [1]) is commonly used for internet-technical specifications. Each of the enhanced forms can be converted into the basic BNF.

Figure 3: Two ways to define the abstract syntax

The abstract syntax for each context-free grammar $G$, denoted by $AST(G)$, can be constructed in two ways (Fig. 3). The first way, originating from logic and more common in the area of compiler construction and verification, is by a signature $\Sigma(G) = (N, CO)$, where $N$ is now seen as a finite set of sort symbols, and $CO$ is a set of constructors with $CO = \{ e_p : X_1 \times \ldots \times X_n \rightarrow X \mid \exists : p = (X \rightarrow w_1X_1w_2 \ldots w_nX_nw_{n+1}) \in P, w_i \in T^*, 1 \leq i \leq n+1, X_j \in N, 1 \leq j \leq n \}$. The set of all ground terms $T_{\Sigma(G)}$ is the abstract syntax of $L(G)$. Because $T_{\Sigma(G)}$ results from $\Sigma(G)$, and $\Sigma(G)$ can be uniquely constructed from $G$, we use the term $G$-algebra instead of $\Sigma(G)$-algebra and $T_G$ instead of $T_{\Sigma(G)}$ in the following. The second way, originating from reasoning about formal languages itself and mainly found in the area of complexity and efficiency analysis, is by an abstract grammar $G_{abs}$, which is a (linear) tree grammar. The language $L(G_{abs})$ is the abstract syntax of $L(G)$. It can be shown that $T_G$ is equal to $L(G_{abs})$.

Figure 4: Two ways to define $L(G)$

The language of a context-free grammar $G = (N, T, P, S)$, denoted by $L(G)$, can be constructed in two ways (Fig. 4). The first is via derivation from the start symbol in $G$, $L(G) = \{ w \in T^* \mid S \Rightarrow^*_G w \}$. The second is via a homomorphism $eval_{Word}$, which maps each syntax tree to a string of $T^*$ in the following way. Each constructor $co_{Xw_0X_1w_1 \ldots X_nw_n} \in \Sigma(G)$, that is, the constructor resulting from the production rule $X \rightarrow w_0X_1w_1 \ldots X_nw_n \in P$ with $w_0, \ldots, w_n \in Z^*$ and $X_1, \ldots, X_n \in N$, is interpreted as $co_{Word(G)}^{Word(G)} : T^* \times \ldots \times T^* \rightarrow T^*$ with $(v_1, \ldots, v_n) \mapsto w_0v_1w_1 \ldots v_nw_n$ for all $v_1, \ldots, v_n \in T^*$. Let $L(T_G) = eval_{Word(G)}(T_G)$, then the resulting tuple $(T^*, L(T_G))$ is a $G$-algebra, named word algebra, and $L(G)$ is the subset of $L(T_G)$ that only contains strings resulting from syntax trees of sort $S$. Fig. 5 gives an overview. Areas of concrete syntax are highlighted in the figures by a red background, and areas of abstract syntax by a green background.\(^3\)

\(^3\)The notation stems from SeeMe [12] and is mainly used in knowledge management. Blue-filled rectangles represent entities, and yellow-filled rectangles with rounded corners represent functions, which are called activities there.
3 Algebraic Framework for Hyperdocument Systems

This new framework for hyperdocuments and hyperdocument systems is restricted to hyperdocuments that can be described with markup languages, amalgamates central aspects of existing hyperdocument models, and strips off a lot of model-specific ballast and proprietary notation.

Markup languages can be characterized via particular context-free grammars, so-called XML grammars. Let $G(SP') = \left((N \cup ˜N) \cup (AT \cup ˜AT) \cup S', T \cup A', P, S\right)$ be a grammar over a basic specification. $G(SP')$ is an XML grammar if all production rules $p$ are either of the form $X - \rightarrow \langle t \ ˜Y \rangle \ (\ ˜X_1 | \ldots | \ ˜X_n) \langle /t \rangle$ for $X \in N$ or $X \rightarrow u = "v"$ with $u \in String, v \in A'$ for $X \in AT$ or $X \rightarrow \epsilon$ for $X \in ˜N \cup ˜AT$. Converting the rule for $X \in N$ from the shorter EBNF into a basic BNF notation results in $n$ rules, where each rule $p_i, 1 \leq i \leq n$, has the form $X \rightarrow \langle t \ Y \rangle \ X_i \langle /t \rangle$. With the previously shown abstraction mechanism, this leads to a constructor $c_{p_i} : Y \times \ X_i \rightarrow X$. As the terminal symbol $t$ provides us with a unique identifier, we name the constructor $t_{X_i}$ instead of $c_{p_i}$. Each rule $p$ of the form $Y \rightarrow u = "v"$ with $u \in String, v \in S'$ leads to a constructor $c_p : String \times S' \rightarrow Y$, which we name $av_Y$ instead of $c_p$. Therefore, an abstract syntax and a validating parser, which reads a correct document in a particular markup language into its abstract representation, can be generated automatically, and a hyperdocument can be defined via a variable-free term of a constructor-based signature. This is the first part of the hyperdocument system depicted on the left-hand side of Fig. 5.

We now have different possibilities to interpret or evaluate the syntax tree into a view, depicted on the right-hand side of Fig. 5. A view in this context is every non-textual representation, most often a graphically rendered output on a screen. A language that describes views is called layout language, and the graphical representation is created by a rendering engine. We can identify at least three typical kinds of views. First, the document view on the screen, well known from each browser. Second, the document tree view as defined by the W3C and often used for theoretical examination of hyper structures.

---

4 This is an attributed extension of the XML grammars introduced by [4] or the similar balanced grammars of [6].

5 String is assumed to be a sort in each basic specification.

6 How rendering engines work in detail is beyond the scope of this work.
A *document tree* is an unranked, sibling-ordered, labeled tree that has two different kinds of leaves, called *content nodes* and *attribute nodes*. And third, the *site map view* that focuses on the links and relations between documents or parts of documents. Additionally, output for special devices, such as screen readers or Braille terminals, can be simply seen as another algebraic interpretation. Therefore, browsers can also be classified by different algebras.

A real-world hyperdocument usually does not consist of a single word of a markup language, but is a bundle consisting of a *schema*, written in a *schema language S*, a *document description*, written in a *markup language ML(S)* that depends on the schema, and a *stylesheet*, written in a *stylesheet language SL*, depicted by the entity at bottom left in Fig. 7.

The most popular schema languages are the document-based XML Schema (c.f. [30]), the Document Type Definition (DTD), and the pattern-based RelaxNG (c.f. [23]), where each DTD language definition can easily be transformed into an XML Schema language definition.

*Document-based* means that a schema S is in principle a tree grammar, coded in a meta syntax similar to EBNF notation. This tree grammar does not directly describe the markup language we want to define, but is the abstract syntax for this markup language. The markup language itself is a particular interpretation, named the *ML(S)*-*algebra*, of the abstract syntax defined by the schema. Moreover, the schema defines additional attributes and some constraints, e.g. for the number of times an element can occur. For the simple special cases that it can occur either zero or one times, exactly once or infinitely often, we can find a context-free representation, but characterizing a range in between is not possible, at least not with reasonable effort. Also, the fact that an attribute is required and that it has a default value cannot be captured by a context-free grammar. So we need an additional constraint base that
does not influence the syntax of the markup language but influences the parser. The parser must reject badly-formed documents, but also documents that do not fulfill the additional constraints given by the schema.

*Pattern-based* means that the set of document trees, not that of syntax trees, is characterized via allowed patterns of the paths of the document trees. In the framework presented, pattern-based approaches are harder to handle. The reason is that there is no canonical way to build an abstract syntax for the markup language for the algebra of documents trees. So usually one has to find a morphism between the document tree algebra and the markup language algebra. Each document-based schema can be translated into a pattern-based one ([18]), and a subset of RelaxNG, called BonXai, [18], can be translated into a document-based form.

![Figure 8: Schema and stylesheet languages](image)

As a running example, we use MiniGPX, a shortened version of the GPS Exchange Format for exchanging geodata[^7]. Though this is not a typical markup language for hyperdocuments, it has a comparatively short and understandable schema definition[^8] and nearly all features can be demonstrated with this language.

All elements and types of the schema are represented in the abstract syntax as constructors. Attributes that belong to complex types are represented as constructors of the corresponding attribute sort. Constraints, such as minOccurs or maxOccurs, have the default value 1. This means that e.g. metadata should appear either exactly once or not at all. Basic types, such as xsd:string, are assumed to have a predefined interpretation.

```xml
<xsd:element name="gpx" type="gpxType"/>
<xsd:complexType name="gpxType">
  <xsd:sequence>
    <xsd:element name="metadata" type="metadataType" minOccurs="0"/>
    <xsd:element name="wpt" type="wptType" minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="trk" type="trkType" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="version" type="xsd:string" use="required" fixed="MiniGPX"/>
  <xsd:attribute name="creator" type="xsd:string" use="required"/>
</xsd:complexType>
```

[^7]: http://www.topografix.com/gpx.asp
[^8]: http://www.topografix.com/gpx/1/1/gpx.xsd
This complex type from the schema $S_{GPX}$ given above can be represented by a signature $\Sigma(S_{GPX}) = (N, OP)$ with

$$OP = \{ \text{root} :: \text{gpxType_av} \times \text{gpxType} \to \text{gpx}$$
$$\text{gpxType} :: \text{metadataAlt} \times \text{wptList} \times \text{trkList} \to \text{gpxType}$$
$$\text{gpxMiniGPX} :: \to \text{gpxType_av}$$
$$\text{gpx_creator} :: \text{xsd:string} \to \text{gpxType_av}$$
$$[::] :: \to \text{gpxType_avList} \ldots \}$$

To show that the proposed framework works reasonably well, we have prototypically implemented some core features in the functional programming language Haskell. Both markup languages and functional programming languages are declarative, but programming languages are better suited for structuring problems and building abstractions than markup languages. The goal of structural markup, where documents are specified in terms of their logical features rather than of particular rendering procedures, is similar to the ideals of functional programming, where computations are specified in mathematical rather than machine-oriented terms. Hyperdocuments can be seen as trees, and functional languages usually offer extensive facilities for representing and manipulating trees. Moreover, if a typed functional language is used, the type system can provide additional structure and integrity.

To make the gap between the concept and the implementation as large as necessary and as small as possible, a signature is implemented as a polymorph data type. Let $\Sigma = (S, OP)$ be a signature with $S = \{ s_1, \ldots, s_n \}$ and $OP = \{ f_{11} : w_{11} \to s_1, \ldots, f_{1n_1} : w_{1n_1} \to s_1, \ldots, f_{k1} : w_{k1} \to s_k, \ldots, f_{kn_k} : w_{kn_k} \to s_k \}$. Each sort is now implemented as a type variable, and each function symbol is implemented as an attribute. Let $(f : \epsilon \to s)' = \text{def } f :: S$ and $(f : s_1 \ldots s_n \to s)' = \text{def } f :: s_1 \to \ldots \to s_n \to s$. We get the following data type:

```
data SIG s_1 \ldots s_k = SIG \{(f_{11} :: w_{11} \to s_1)', \ldots, (f_{1n_1} :: w_{1n_1} \to s_1)', \ldots, (f_{k1} :: w_{k1} \to s_k)', \ldots, (f_{kn_k} :: w_{kn_k} \to s_k)' \}
```

The signature $\Sigma(S_{GPX})$ is implemented by the following data type:

```
data GpxSIG gpx gpxType gpxType_av \ldots =
GpxSIG {rootMt :: [gpxType_av] -> gpx,
        root_ :: [gpxType_av] -> gpxType -> gpx,
        gpxType_ :: (Maybe metadata) -> [wpt] -> [trk] -> gpxType,
        gpxMiniGPX :: gpxType_av,
        gpx_creator :: XSD_string -> gpxType_av, \ldots}
```

A framework is called schema-aware if the data binding and the processing, manipulating or generation of documents depend on a given schema. E.g. the special purpose functional languages XSLT and FXT, implemented in SML, transform arbitrary document trees independently of the schema, and so they need no implementation of XML types. The Web Authoring System Haskell (WASH) represents the XHTML schema as a Haskell data type. HaXml gives a translation of DTDs to Haskell types, and UUXML gives a type-preserving XML Schema/Haskell data binding. As far as we know, all schema-aware approaches for hyperdocuments embed XML values by finding a suitable

---

10 This implementation technique is proposed by [26] in the context of compiler construction and is still under development.

11 [http://www.w3.org/TR/xslt](http://www.w3.org/TR/xslt)

---

39
translation between XML types, described by schemas and types of the programming language. Our implementation first translates schemas into constructor signatures and then uses a translation between constructor signatures and Haskell types.

Each algebra is then implemented by simply instantiating the type variables with the concrete types of the carrier sets, and each attribute is instantiated by the corresponding function of the given algebra. So the algebra that interprets a document as an abstract syntax tree of $AST_{ML(S)}$ looks like this in the Haskell notation:

```haskell
gpxALG :: GpxSIG Gpx GpxType GpxType_av ...
gpxALG = GpxSIG Root_Mt Root_ GpxType_ Gpx_MiniGPX Gpx_creator ...

data Gpx = Root_Mt [GpxType_av] | Root_ [GpxType_av] GpxType
data GpxType = GpxType_' (Maybe Metadata') [Wpt] [Trk]
data GpxType_av = Gpx_MiniGPX | Gpx_creator XSD_string
```

A second algebra, which interprets a document as an object of the Scalable Vector Graphics (SVG)\(^{12}\), can look like this:

```haskell
svgALG = GpxSIG gpx_ gpxType_ gpxType_av_ ... =
  where root_mt_ = "<svg />
    root_ avlist gpxType = "<svg xmlns="http://www.w3.org/2000/svg"
      xmlns:svg="http://www.w3.org/2000/svg"
      xmlns:xlink="http://www.w3.org/1999/xlink">"
      ++ gpxType ++ "</svg>"
    gpxType_ metadata wpt trk = "<g>" ++ (conc wpt) ++
      (conc trk) ++ "</g>"
    gpx_MiniGPX = "MiniGPX"
    gpx_creator c = c
```

The *stylesheet* is the formatter for a hyperdocument. It defines path expressions to locate a particular place in the document tree, and actions that modify the selected part of the tree. In practice, a stylesheet is usually described by CSS (c.f. \([29]\)) or XSLT\(^{13}\). CSS stylesheets can only modify values of attributes, but not the document tree itself. XSLT is a tree-transformation language that can also change the structure of the tree. Because the hyperdocument is represented by a syntax tree $doc \in AST_{ML(S)}$, the stylesheet must be interpreted by paths on $doc$, and the action by tree transformations at the located place. After executing the actions, we get a modified syntax tree, which is then interpreted by a layout language, depicted by the entity at bottom right in Fig. 7.

The parser that can be generated from the grammar can be realized in Haskell in a monadic style. It not only parses the document itself, but is parameterized, so that it can be used to compile a source document directly into a target interpretation.

```haskell
parseGpx :: GpxSIG gpx gpxType gpxType_av metadata metadataType wpt wptType
           wptType_av trk trkType name time bounds boundsType_av ele
           sym number trkseg trksegType trkpt -> MParser Char gpx
parseGpx alg = do result <- (parseE 'parM' parseC); return result
```

\(^{12}\)http://www.w3.org/Graphics/SVG/
\(^{13}\)http://www.w3.org/TR/xslt
where parseE = do avlist <- (opencloseAV "gpx" (parseGpxType_av alg));
               return (root_mt alg avlist)
parseC = do avlist <- (openAV "gpx" (parseGpxType_av alg))
            content <- (parseGpxType alg)
            close "gpx"
            return (root_ alg avlist content)

parseGpxType :: ... -> MParser Char gpxType
parseGpxType alg = do result1 <- qmM' (parseMetadata alg)
                      result2 <- starM (parseWpt alg)
                      result3 <- starM (parseTrk alg)
                      return (gpxType_ alg result1 result2 result3)

parseGpxType_av :: ... -> MParser Char gpxType_av
parseGpxType_av alg = parL [parseGpx_MiniGPX alg, parseGpx_creator alg]
        where parseGpx_MiniGPX alg = do isTag "version"
                              isChar '='
                              result <- parseDQ
                              return (gpx_MiniGPX alg)
parseGpx_creator alg = do isTag "creator"
                          isChar '='
                          result <- parseDQ
                          return (gpx_creator alg result)

If parseGPX is called with gpxALG and applied to a document from $L(ML(S))$, then the result is a syntax tree from $AST_{ML(S)}$.

<gpx version="MiniGPX" creator="JOSM GPX export">
  <metadata>
    <bounds minlat="51.4813" minlon="7.3855"
           maxlat="51.5019" maxlon="7.4255"/>
  </metadata>
  <trk>
    <name>H-Bahn</name>
    <trkseg>
      <trkpt lat="51.4921" lon="7.4166">
        <time>2008-03-28T17:02:06Z</time>
      </trkpt>
      <trkpt lat="51.4922" lon="7.4167">
        <time>2008-03-31T22:29:55Z</time>
      </trkpt>
      <trkpt lat="51.4922" lon="7.4168">
        <time>2008-11-27T12:47:33Z</time>
      </trkpt>
    </trkseg>
  </trk>
</gpx>
Specifying Hyperdocuments with Algebraic Methods

Mattick

Root_ [Gpx_MiniGPX,Gpx_creator "JOSM GPX export"]
  (GpxType_
    (Just (Metadata_ (MetadataType_
        Nothing
        Nothing
        (Just (Bounds_mt [Bounds_minlat 51.4813,Bounds_minlon 7.3855,
            Bounds_maxlat 51.5019,Bounds_maxlon 7.4255])))))
  []
  []
  [Trk_ (TrkType_ (Just (Name_ "H-Bahn"))
    [Trkseg_ (TrksegType_ [
      Trkpt_ [WptType_lat 51.4921,WptType_lon 7.4166]
        (WptType_ Nothing (Just (Time_ "2008-03-28T17:02:06Z")) Nothing Nothing),
      Trkpt_ [WptType_lat 51.4922,WptType_lon 7.4167]
        (WptType_ Nothing (Just (Time_ "2008-03-31T22:29:55Z")) Nothing Nothing),
      Trkpt_ [WptType_lat 51.4922,WptType_lon 7.4168]
        (WptType_ Nothing (Just (Time_ "2008-11-27T12:47:33Z")) Nothing Nothing)])])
  ]

If it is called with svgALG, the result is an SVG term:

<svg>
  <g>
    <line fill="none" stroke="#000000" stroke-width="2" x1="51.4921"
      x2="51.4922" y1="7.4166" y2="7.4167"/>
    <line fill="none" stroke="#000000" stroke-width="2" x1="51.4922"
      x2="51.4922" y1="7.4167" y2="7.4168"/>
  </g>
</svg>

4 Hyperdocument Engineering

Hyperdocument engineering can benefit from this framework in multiple ways, e.g. in the areas of adaptable hyperdocuments, cf. [8], and universal design. There, it must be possible to design documents that can be to a great extent tailored to the reader’s needs and wishes. So it would be a great help for the developer to have a mechanism for specifying a set of documents for which the specified aspects are fixed, and the rest is open to a user’s adaption. This can be done by using syntax trees with variables as an abstract representation of so-called semi documents [19]. Given a markup language ML(S) and assuming that the interpretation of AST_{ML(S)} is known, a syntax tree with variables describes a subset of AST_{ML(S)}, whereas a syntax tree without variables describes a single element of it. Of course, in the end, the hyperdocument system must deliver only a single element and not a set of elements to the rendering engine. So, in addition, we need a user profile that characterizes the documents the user allows. In contrast to developers, users more often think in terms of views. To describe profiles, we use a particular profile description language P that can specify views in the way schemas describe documents (cf. the bottom right part of Fig. 9). Now the adaptable hyperdocument system must search for an assignment of the variables with which the interpretation of the corresponding variable-free syntax tree fits the constraints of the user profile. This can also be a first step towards a hyperdocument description language that specifies documents in a screen-oriented setting, which is useful because producers and consumers are increasingly the same persons.
Instead of obtaining the necessary information from a user profile, it is also possible to collect design goals in a requirements engineering phase. We then enrich the signature $\Sigma = (S, F)$ with suitable relations that we need to express the requirements. With $\Sigma = (F, S, R)$, it is possible to capture the requirements by axioms $AX$, and the specification $SP = (\Sigma, AX)$ specifies a set of allowed algebras. With this technique, it is possible to semi-automatically generate documents out of semi documents.

Because the markup language is an algebra of the abstract syntax, it is possible to use syntax trees or a language that specifies syntax trees as an executable specification language for hyperdocuments (cf. [20]). As in programming languages, it is easier to test constraints and requirements on the abstract level than on the level of the concrete representation. The concrete hyperdocument is then just an interpretation with the $ML(S)$ algebra.

5 Conclusion and Future Work

The algebraic approach presented gives a precise and clear model for understanding XML-based hyperdocuments and hyperdocument processing. It shows new techniques for hyperdocument systems, especially browsers, resulting from research in compiler construction. Known techniques and results from this area can be adapted. The abstract syntax trees are used to store XML-based hyperdocuments unambiguously, a great advantage over document trees. It is demonstrated how real-world hyperdocuments, described by a schema, a document description, and a stylesheet, can be handled by our approach. Benefits for hyperdocument engineering are only sketched, but not elaborated to their full potential. For example, it has not yet been examined how these techniques can improve hyperdocument editors, where the source documents are specified via a layout language, and the document structure in a markup language is the target. The functionality of the core elements of our framework is proven by prototypical Haskell implementations, and it shows the usability of the model, but it is far from being a software system that can be used in real-world hyperdocument engineering. To exhaust the full potential of the framework, the term-rewriting language Maude [7] could be an interesting alternative to Haskell.

References


Reconstructing Information Retrieved from Multiple Websites

Héctor Valero
Universitat Politècnica de València
València, Spain
hecvalli@posgrado.upv.es

Carlos Castillo
Universitat Politècnica de València
València, Spain
carcasg1@posgrado.upv.es

Josep Silva
Universitat Politècnica de València
València, Spain
jsilva@dsic.upv.es

Abstract

This work presents a model for information retrieval from multiple webpages. This model does not need to pre-process, parser or label the webpages; and thus, it can work online and in real time. The model introduces two new techniques for visualization that allow us to automatically reconstruct a new webpage with the information retrieved. This page is structured taking into account the semantically related information. The technique has been implemented and the implementation is discussed focussing on the main problems that appear when the proposed algorithms are integrated into a commercial web browser.

Keywords: Information retrieval, HTML filtering, Webpages visualization.

1 Introduction

Currently, information retrieval is one of the hot topics in Internet; and indeed more in the semantic web. However, the lack of online and real time applications able to retrieve information automatically, is a sign of the difficulties of this task. Current techniques for information retrieval in Internet are mainly specialized in retrieving webpages that are related to a particular query \[1\]. In this context, search engines such as Google or Bing implement very accurate and precise algorithms for finding the webpages related to a given query. Nevertheless, in many cases, there is too much information contained in the webpage that is not related to the user’s query. Thus, the granularity level of the answer is too big: a whole webpage.

In the context of the semantic web, it is frequent to produce more concrete results that consist of texts that answer a given question. However, these techniques need to pre-process the web pages that are used as the sources of information. A common approach is to build an ontological model that is constructed and queried with languages such as RDF \[2\] and OWL \[3\]. This imposes important restrictions on the webpages that are going to be processed; and for this reason, the tools implemented with this approaches are often offline. The tools that use microformats \[4,5,6\] have the same problem. In particular, webpages that use microformats could be automatically processed thanks to the use of special meta-labels that qualify the information and that are inserted in the (X)HTML code. But, unfortunately, the technology that supports microformats is not mature enough, and this is the reason why much of the webpages in Internet do not use microformats. Therefore, they cannot be processed in real time.

In a previous work \[7\], we proposed an online and real time technique for information retrieval that is able to automatically retrieve information related to a given query from a single webpage. This technique

\[\text{\textit{This work has been partially supported by the Spanish Ministerio de Ciencia e Innovación under grant TIN2008-06622-C03-02, by the Generalitat Valenciana under grant ACOMP/2009/017, and by the Universidad Politécnica de Valencia (Program PAID-06-08).}}\]
is based on the use of syntax distances as a measure of semantic relations. Roughly, the technique extracts from a webpage those elements that are syntactically close to the terms specified by the user. Therefore, the technique assumes that those terms that are syntactically close are more semantically related. The technique was implemented and integrated into the Firefox web browser after it was approved by the Firefox experts developers area. Its extensive use with hundreds of users has confirmed that the use of syntax distances is a simple but powerful idea that works in practice [8].

Recently, we have developed a new information retrieval technique that generalizes the idea of using syntax distances between elements of a same page to multiple webpages incorporating page distances, domain distances, etc. This new distance has been named hyper-syntactic distance. The main problem of the new technique is the difficulty of automatically reconstructing a new webpage with the final result. In particular, when working with different webpages, new visualization problems appear such as the overlapping of elements with absolute positions, the incompatibility of different layouts, the integration of CSS files; and others such as the security imposed by the browsers, the time needed to load several webpages, etc.

In this work we face these problems and explain different approaches to solve them. When retrieving information from multiple webpages, we find that showing the information to the user is not as simple as joining together all the blocks of information retrieved from the webpages. The information must be presented in a way that the interrelations between the information from different webpages is explicit. In order to do this, we propose two visualization models: the tabular model and the hierarchical model.

The main advantages of this new technique are that it does not need the use of proxies [9], it can work online and in real time—with any webpage—without any pre-compilation or pre-processing phase [10]; it can process multiple sources; and it can retrieve information with a low granularity level: one single word.

The rest of the paper has been organized as follows. Section 2 presents our technique by using a motivating example. Section 3 explains a novel technique for information retrieval from multiple webpages. This technique uses two visualization models that are introduced in Section 4. In Section 5, we describe our implementation. The main problems are discussed and their solutions explained. This section also presents the results obtained by a performance evaluation of our tool. Finally, Section 6 concludes.

2 Motivation

This section presents a real example of information retrieval using the technique presented in this work. Let us consider a user that is browsing in Internet and she loads the main webpage of the United States Department of Labor searching for work.

In a normal scenario, the user reads the main webpage of the department (see Figure 1) and, using the standard browser’s filter or the search box of the webpage (if it is provided), she looks for the word searched in the page, and she loads a new page using the corresponding hyperlink. If the information searched is found, the process finishes, but if the information is not in the loaded webpage, the user has to return to the previous page and explore another hyperlink. This is repeated successively until the desired information is found. During this process, (i) the user is forced to read much information not related to what she is looking for. This information appears in the webpages visited during the search, and (ii) she must explore several hyperlinks until the relevant hyperlinks are found. This process is a time-consuming task.

In a second scenario, the user uses our tool for information retrieval. She loads the main webpage of the United States Department of Labor, she types the filtering criterion (“work”) and she clicks on the filtering button. With a simple click, the tool automatically constructs a new webpage with all the
information related to work in the main webpage and in the webpages accessible from it. During the analysis of the webpages, the tool explores the relevant hyperlinks to extract the desired information from each of them. With all the information retrieved, a new webpage with the results is generated. This webpage, only contains information relevant for the user.

In Figure 2 all the occurrences of the filtering criterion are highlighted. This is the result of using...
the standard search with the term “work”. Unfortunately, the information provided by this search is poor; but it includes some useful hyperlinks such as “work hours” (at the left). Our algorithm is able to automatically filter this webpage and explore each relevant hyperlink to reach other pages in the same or in other domains in order to gather all the relevant information according to the filtering criterion. With this information, the algorithm produces a new webpage as the one shown in Figure 3.

In this case, the model used to reconstruct the final webpage is the **hierarchical model**. The hierarchical model groups the retrieved information blocks according to their minimum syntactical distance; placing them as close as possible. For this representation, the information blocks are nested immediately after the hyperlink that points to the webpage to which they belong. This process is recursively repeated with the new hyperlinks that are found inside these retrieved blocks.

In the example, the first information shown is composed of the blocks retrieved from the main webpage of the Department of Labor. From these blocks the hyperlinks are extracted and ordered by relevance according to their syntactical distance. Then, these new webpages are analyzed producing new relevant blocks that are placed next to their corresponding hyperlinks in the already visualized webpage. Internally, a block with relevant information is a set of HTML nodes (i.e., a DOM subtree of an HTML webpage) grouped inside an HTML container (e.g., a table, a layer, etc.). In Figure 3 we can observe that some blocks are placed after their corresponding hyperlink. For instance, at the top, we have a block after the “work hours” hyperlink. This block has been extracted from the webpage that contains information about work hours regulations in USA. The original webpage about work hours is shown in Figure 4. Note that the results webpage only shows a part of the information of this webpage. The blocks of information extracted contain the information that is related to the filtering criterion. The other information was discarded.

In Figure 3 the third block of information contains information related to worker’s compensation. This block is placed after the block associated to work hours because it has been extracted from the page pointed by the hyperlink after the one that pointed to the work hours webpage. In particular, the webpage source of this information is shown in Figure 5. Some blocks are nested in the final webpage.
As a consequence of this scheme, in this model the webpages are decomposed and their relevant blocks are mixed in a hierarchical shape according to their navigational relations.

In addition to the hierarchical model, in this work we propose another model called the tabular model. In contrast to the hierarchical model, the tabular model provides a representation with a page granularity.
That is, the information retrieval engine retrieves blocks with relevant information from each analyzed webpage; but these blocks are maintained in the final representation so that they keep their original structure. Both models will be presented in the next sections.

3 Information Retrieval from Multiple Webpages

This section presents an information retrieval algorithm able to extract information from multiple webpages. Our algorithm uses a previous algorithm to extract information from single webpages. The details of this algorithm can be found in [11]. In the following, we will assume the existence of a function getSlice(p,q) that implements this algorithm. Given a webpage p and a filtering criterion q, getSlice extracts from p all the information related to q.

We provide now a precise definition of webpage and filtering criterion. We will assume that a webpage is represented by a DOM tree [12].

Definition 3.1 (DOM tree). A DOM tree t=(V,E) is a tree whose nodes V are labeled with HTML labels, and they interconnected with a set of edges E according to the DOM specification [12].

Definition 3.2 (webpage). A webpage is a tuple (u,t) where u is a URL and t is a DOM tree.

A filtering criterion is composed of one or more words associated to the information that we want to retrieve; and one proximity measure that represents the syntax distance between what we are looking for and what we retrieve.

Definition 3.3 (Filtering criterion). A filtering criterion is a pair (w,d) where w is a string that represents the desired information; and d is an integer that represents the precision used in the search.

Example 1. The webpage in Figure 6 (left) is the main webpage of the Technical University of Valencia. If we filter this webpage with the filtering criterion (“student”, 0), we get the webpage in Figure 6 (right).

Figure 6: Webpage of the Technical University of Valencia (left) and its filtered version (right)

The DOM tree of this webpage is huge. Hence, we focus on a part of the webpage. Concretely, if we observe the text in the gray column at the left, the part of the DOM tree that represents this text is shown in Figure 7.

In the figure, the black node contains the word “student”. From this node, all its ancestors and successors are kept in the filtered webpage. Because the required precision is 0, no more nodes are retrieved, and thus, white nodes are discarded. The produced subtree corresponds to the text in the gray area of the filtered webpage.

---

1This is the English version, but note that, since the technique is based on syntax distances, it works equally with any language.
3.1 The Hyper-syntactic Distance

Usually, there exist in a webpage many hyperlinks to other related webpages. An information retrieval algorithm should be able to explore all these hyperlinks in order to reach the relevant information of other webpages and then, reconstruct a new webpage with all the retrieved information. However, we want our technique to work online and in real time. This means that loading all the hyperlinks is not possible because it would require too much time. Therefore, our technique must explore only some hyperlinks during the search and discard the others. In order to solve this problem, in a previous work [13] we proposed a unit of measurement called hyper-syntactic distance. It allows us to order the hyperlinks of a set of webpages by relevance with respect to a filtering criterion. In essence, the hyper-syntactic distance determines the relevance of a hyperlink $H$ in a webpage $p$ with respect to a DOM node $n$ in a webpage $p'$ considering three fundamental measures: the distance in the DOM tree from $H$ to $n$ if $p = p'$, or from $H$ to the root of $p$ if $p \neq p'$; the number of pages that must be traversed from $p'$ to $p$ and the number of domains that must be traversed from the domain of $p'$ to the domain of $p$. The combination of these measures produces a value that is known as hyper-syntactic distance and that approximates the semantic relation $la$ between $n$ and $H$.

In the following, we will assume the existence of a function $getMostRelevantLink(links, q)$ that given a set of hyperlinks $links$, and a filtering criterion $q$, it returns the hyperlink that is more relevant considering its hyper-syntactic distance with respect to $q$. The interested reader is referred to [13] where this unit of measurement is explained in detail.

For our purposes, in the rest of the article we can view an hyperlink as a directed arc between two webpages.

**Definition 3.4** (Hyperlink). An hyperlink is a pair $(u, v)$ where $u$ is the URL of a source webpage and $v$ is the URL of the target webpage.

Thanks to the hyper-syntactic distance, we could define a simple information retrieval algorithm that repeats three fundamental steps:

1. Filter the current webpage ($getSlice$)

2. Find the most relevant hyperlink in the loaded webpages ($getMostRelevantLink$)
3. Load the webpage pointed by the most relevant hyperlink

Finally, when no more relevant hyperlinks exist, we could reconstruct a new webpage with the retrieved information. Nevertheless, this scheme is not appropriate for an online tool. The reason is that loading a webpage needs approximately one second. This means that the previous scheme would not show to the user any result until all the webpages had been analyzed and filtered, that implies too much time for a real time tool. Remember that web design guidelines establish 10 seconds as the maximum response time \[14\]. Therefore, we propose a more appropriate solution in which the information is reconstructed and shown to the user incrementally as it is being retrieved. This means that the user can see the retrieved information from the first second, and this information is increased as the analysis continues.

4 Visualization of the Retrieved Information

This section introduces a new technique to incrementally integrate and visualize the information recovered from multiple webpages. This technique uses two independent (but very related) algorithms for the visualization of the information. The first one presents the information tabularly, the second one uses a hierarchical representation.

Both algorithms are able to retrieve information from different webpages and show it incrementally while it is being recovered. The main difference between them is the way in which the information is visualized in the browser.

**Tabular Visualization** The lowest granularity level in this representation is a page. Basically, the final webpage is a linear succession of the filtered webpages. Each filtered webpage is considered as a whole, and thus, all the information that appeared together in the filtered webpage, is also together in the final webpage. The filtered webpages are ordered according to their navigational structure using a depth-first order.

**Example 2.** The next figure shows a set of linked webpages where the dark part represents the relevant information. At the right, we see the tabular representation of this relevant information.

![Tabular Representation](Image)

**Hierarchical Visualization** The lowest granularity level in this representation is a word. In this representation, the final webpage is a tree where the filtered webpages are organized. In contrast to the tabular representation, the filtered webpages can be mixed because each filtered webpage is placed next to the hyperlink that references it.

**Example 3.** The following figure complements Example 2 showing the hierarchical representation of the same set of webpages.

![Hierarchical Representation](Image)
4.1 Tabular Visualization

Algorithm 1 implements the tabular visualization model. This algorithm uses the following functions:

timeout() This function controls that the algorithm is not executed more time than the specified by the user in the configuration. When the specified time is reached it returns True.

getSlice(p, q) It returns the slice produced after filtering webpage p with the query q. This is done using the algorithm proposed in [11].

createIframe(d) This function creates and returns one DOM node of type iframe whose content is the DOM tree d received as a parameter.

append(n, m) It appends the DOM node m as a child of the DOM node n.

getLinks(nodes) It extracts all the hyperlinks of a set nodes of DOM nodes.

getMostRelevantLink(links, q) It extracts from the set links the hyperlink with a lower hyper-syntactic distance with respect to the filtering criterion q. It is used to determine what is the next hyperlink that should be processed.

load(page) It loads the webpage page.

getNode(l) It returns the DOM node associated to hyperlink l.

Function processWebPage is a recursive function that loads the most relevant pages from a set of hyperlinks that are potentially processable. Each time a new webpage is loaded, it is analyzed and new relevant hyperlinks are added to the set. Every loaded webpage is parsed, then filtered, and the result produced is shown with function show. This implies that the final webpage is shown incrementally, page after page. This process is repeated until a timeout is reached.

Function show has been specialized for each model. In the tabular model, it creates an iframe whose content is the webpage that has been just filtered, and this iframe is put next to the webpage that contains the hyperlink that pointed to this webpage.

4.2 Hierarchical Visualization

Algorithm 2 implements the hierarchical visualization model. This algorithm, shares most of the functions (including processWebPage) used in Algorithm 1 that were explained in the previous section. In addition, it uses the following functions:

getParent(n) It returns the first ancestor of node n that is of type container (table, div, frame, etc.).

createContainer() It creates and returns a DOM node of type table.
Algorithm 1 Tabular Visualization

**Input:** A webpage $P$, and a filtering criterion $q$

**Output:** A webpage $P'$

**Initialization:** $link = \emptyset$

function $show(Node n, DOM d)$

$showTabular(n, d)$

function $showTabular(Node n, DOM d)$

$iframe = createIframe(d)$

$append(n, iframe)$

function $processWebPage(Link l, WebPage p)$

$relevantNodes = getSlice(p, q)$

if ($l \neq \emptyset$)

then $nodeC = getNode(l)$

else $nodeC = \langle BODY \rangle$

$show(nodeC, relevantNodes)$

$links = links \cup getLinks(relevantNodes)$

if ($timeout() \lor links = \emptyset$)

then exit()

else $link = getMostRelevantLink(links, q)$

$links = links \setminus link$

$newPage = load(URL2)$ where $link = (URL1, URL2)$

$processWebPage(link, newPage)$

**return**

$< HTML >$

$< BODY >$

$processWebPage(link, P)$

$< \langle HTML \rangle$

$< \langle BODY \rangle$

The behavior of this algorithm and Algorithm I is very similar. The main difference is function $show$. In this case, $show$ has been specialized to show the filtered webpage as a part of the webpage that referenced it. This is done by creating a new container of type table. The type table is a good selection because it allows us to establish relative sizes and because it has a $Z$ axis equal to zero; and thus, it is never superposed to the elements already shown in the page. The new table is linked with the first ancestor node $n$ that is a container. In this way, the new webpage is integrated into the webpage that referenced it, exactly in the point of the webpage where the hyperlink was (in the container of this hyperlink).

5 Implementation

In this section we describe the implementation of the algorithms proposed and we discuss the main problems that emerge when integrating them into a browser.
Algorithm 2: Hierarchical Visualization

**Input:** A webpage \( P \), and a filtering criterion \( q \)

**Output:** A webpage \( P' \)

**Initialization:** \( \text{link} = \emptyset \)

\[
\text{function } \text{show}(\text{Node } n, \text{DOM } d) \\
\quad \text{showHierarchical}(n, d)
\]

\[
\text{function } \text{showHierarchical}(\text{Node } n, \text{DOM } d) \\
\quad \text{container} = \text{createContainer}() \\
\quad \text{if } (n \neq < \text{BODY}> ) \\
\quad \quad \text{then } \text{append}(\text{getParent}(n), \text{container}) \\
\quad \quad \text{else } \text{append}(n, \text{container}) \\
\quad \text{append}(\text{container}, d)
\]

\[
\text{return} \\
\quad < \text{HTML} > \\
\quad < \text{BODY} > \\
\quad \text{processWebPage}(\text{link}, P) \\
\quad < \text{HTML} > \\
\quad < \text{BODY} > \\
\]

The implementation is much more complex than the algorithms presented here because it has to make some transformations of the filtered webpages in order to guarantee that they are correct. For instance, the size attributes of the retrieved webpages must be changed to ensure that they fit into the container where they are inserted. The CSS styles must be imported so that the retrieved information keeps the original format, etc.

All the source code of our tool is open. Therefore, for concrete details about the design decisions taken, we refer the interested reader to:

http://www.dsic.upv.es/~jsilva/webfiltering

The implementation has to perform some additional checks before it loads the webpages pointed by the relevant hyperlinks. In particular, the information retrieval engine only processes (X)HTML pages; hence, it is a waste of time to load files of type PDF, MP3, etc. In order to avoid the load of such files, the object XMLHttpRequest is used to inspect the headers of the page before loading it, and thus, only loading those that contain useful information. For that, we use the following functions:

\[
r = \text{newXMLHttpRequest}(); \\
r.\text{open}(\text{"HEAD"}, \text{url}); \\
r.\text{send}(\text{null}); \\
r.\text{getResponseHeader}(\text{"Content - Type");}
\]

In this section we focus on the two main problems that appear when we implement the algorithms. These problems will appear in any platform or commercial browser where they are implemented:

\footnote{\textit{We are currently implementing an algorithm which is able to extract information from text and PDF documents, but the algorithm which is distributed in Firefox only processes (X)HTML.}}
• **Layers.** One of the most important visualization problems is caused by layers, because they use absolute positions. Concretely, during the filtering phase, it is frequent to find various webpages with layers whose position is defined with absolute values. When this happens, it is possible that, in the final webpage reconstructed from these webpages, the positions of different layers (extracted from different webpages) overlap. An example is shown in Figure 8.

![Figure 8: Problem caused by layers](image)

The solution is to transform all layers with absolute positions to tables of the same size that are placed in the same position than the layer in the original webpage, but they are relocated in the final webpage thanks to the use of relative positions.

• **Security.** Another of the main implementation problems is caused by the security systems of web browsers. In particular, there exists one kind of vulnerability of web browsers called Cross Site Scripting (XSS), where an attacker could execute scripts from a page or domain different from the loaded webpage. XSS has been avoided by current web browsers with a security system that blocks the execution of dangerous code.

As it was explained in previous sections, our algorithms retrieve content from multiple webpages, they filter the content, and finally show a final webpage with the results. Therefore, the final webpage can contain scripts from multiple webpages and domains. When this happens, the security system anti-XSS is activated and it removes all the content that is potentially dangerous. This makes the final result to be incomplete, unstructured, or even completely empty.

The security system anti-XSS works as follows: When a user loads a webpage, the browser explores the DOM tree to find script labels or nodes. If they are found, the script interpreter executes the scripts only if they belong to the loaded webpage. If, contrarily, the script belongs to another webpage, it is blocked. Concretely, the security system is activated when we filter the retrieved webpages; and it blocks all the scripts (and some other insecure elements), removing all the content that could be dangerous.

Our implemented solution is the creation of an iframe object in which we load each filtered webpage. The iframe container does not activate anti-XSS because this container allows the load of URLs, thus embedding webpages inside other webpages. This solution works if we define the
iframe object with some special properties:

```javascript
frame = document.createElement("iframe");
...
frame.setAttribute("type", "content");
...
frame.webNavigation.allowJavascript = false;
frame.webNavigation.allowMetaRedirects = true;
frame.webNavigation.allowPlugins = false;
```

It is interesting to highlight that the iframe must be of type ‘content’. When a container is defined of type content, the browser only draws the information of the webpage but it does not execute any script. For this reason, we can retrieve the filtered information avoiding the anti-XSS security system. When a DOM tree is filtered, we put the new nodes inside a new container and we embed the container in the final webpage.

### 5.1 Performance Evaluation

The main bottleneck of this technique is the load of webpages; because it depends on the connection speed, the current state of the network, and many other external factors that affect the response time. In order to guarantee that the tool is able to work in real time, we must ensure that the results are shown in a bounded time. For this reason, the tool uses function `timeout()` (explained in Section 4.1). According to Jakob Nielsen’s web usability design guidelines [14], in our implementation we established 10 seconds as the default value for the timeout. This value can be changed at any time, but our experiments demonstrate that it is often enough. In Table 1 we show the number of hyperlinks explored, with a timeout of 10 seconds, for several webpages analyzed. The average result is 13.5 webpages analyzed for each URL.

<table>
<thead>
<tr>
<th>URL</th>
<th>Filtering criterion</th>
<th>Links visited</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.iee.org">www.iee.org</a></td>
<td>student</td>
<td>12</td>
</tr>
<tr>
<td><a href="http://www.upv.es">www.upv.es</a></td>
<td>student</td>
<td>18</td>
</tr>
<tr>
<td><a href="http://www.who.int">www.who.int</a></td>
<td>OMS</td>
<td>25</td>
</tr>
<tr>
<td><a href="http://www.un.org">www.un.org</a></td>
<td>Haiti</td>
<td>6</td>
</tr>
<tr>
<td><a href="http://www.esa.int">www.esa.int</a></td>
<td>launch</td>
<td>13</td>
</tr>
<tr>
<td><a href="http://www.nasa.org">www.nasa.org</a></td>
<td>space</td>
<td>16</td>
</tr>
<tr>
<td><a href="http://www.mec.es">www.mec.es</a></td>
<td>beca</td>
<td>18</td>
</tr>
<tr>
<td><a href="http://www.edu.gva.es">www.edu.gva.es</a></td>
<td>universitat</td>
<td>20</td>
</tr>
<tr>
<td><a href="http://www.ilo.org">www.ilo.org</a></td>
<td>projects</td>
<td>8</td>
</tr>
<tr>
<td><a href="http://www.unicef.es">www.unicef.es</a></td>
<td>Haiti</td>
<td>10</td>
</tr>
<tr>
<td><a href="http://www.mityc.es">www.mityc.es</a></td>
<td>turismo</td>
<td>9</td>
</tr>
<tr>
<td><a href="http://www.mozilla.org">www.mozilla.org</a></td>
<td>firefox</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 1: Number of analyzed webpages with timeout=10 seconds

Note that a timeout of 10 seconds is the maximum time used to complete the final results webpage. But the visualization algorithms are incremental, thus, as an average, the first result is shown in less than a second (10/13.5 seconds).

All figures and examples of this paper are real examples produced with analyses made by our tool. More examples and information about the tool can be found at:
6 Conclusions

This article introduces two new models of visualization for information retrieved from multiple webpages. The tabular model is specially good for complex webpages, because it keeps the original internal structure of the loaded webpages, and they are shown as unitary blocks. The hierarchical model is particularly interesting for pages with a lot of textual content, and it works very well, e.g., in forums, where it is able to find the relevant subjects and explore the threads with the answers joining together all the related information.

We are currently studying the possibility of combining both models. This combination would produce a hybrid model able to behave differently depending on the structure of the webpage that has been processed. In addition, we plan to change the hierarchical representation with a new tree-view that represents a personalized website map where the nodes of the tree are the filtered webpages, and where the user could collapse or expand the paths to the information. In this way, the generated web maps would be personalized with the user’s filtering criterion.

References


Incremental Construction of Counterexamples in Model Checking Web Documents
Franz Weitl and Shin Nakajima
National Institute of Informatics (NII)
Tokyo, Japan
Email: {weitl,nkjm}@nii.ac.jp

Abstract
A new algorithm for incrementally generating counterexamples for the temporal description logic ALCCTL is presented. ALCCTL is a decidable combination of the description logic ALC and computation tree logic CTL that is expressive for content- and structure-related properties of web documents being verified by model checking. In the case of a specification violation, existing model checkers provide a single counterexample which may be large and complex. We extend existing algorithms for generating counterexamples in two ways. First, a coarse counterexample is generated initially that can be refined subsequently to the desired level of detail in an incremental manner. Second, the user can choose where and in which way a counterexample is refined. This enables the interactive step-by-step analysis of error scenarios according to the user’s interest.

We demonstrate in a case study on a web-based training document that the proposed approach reveals more errors and explains the cause of errors more precisely than the counterexamples of existing model checkers. In addition, we demonstrate that the proposed algorithm is sufficiently fast to enable smooth interaction even in the case of large documents.

1 Introduction
Model checking is a powerful technique for automatically detecting errors in hard- and software design artifacts that has also been applied to verify business processes [19], web services [15], and web documents [26]. A remaining problem of model checking is its limited usability for non-experts. In the domain of document management, formal verification methods cannot be applied without appropriate user support. In previous work, we proposed means of supporting the user in model generation [23, 22] and property specification [21]. In this paper, we introduce the formal structures and algorithms to support incremental and interactive analysis of model checking results.

A model checker determines if a given finite state transition system $M$ is a model of a temporal formula $p$. If $M$ violates $p$, a counterexample is provided. An ideal counterexample should take a form that demonstrates in a complete yet concise and comprehensible way [6] why $M$ violates $p$. In addition, it should precisely isolate those parts of the model $M$ and the formula $p$ that contribute to a violation. Counterexamples provided by current state-of-the-art model checkers such as NuSMV [5] or SAL [8] consist of finite paths in the state transition system $M$. These “linear” counterexamples are, in general, not complete, i.e., they may demonstrate the cause of the property violation just partially [7]. Even so, they tend to be large and difficult to understand [4, 9]. These problems become worse if temporal formulae contain first order predicates and quantified variables as required, for instance, to express properties for web services [15] and documents [25]. To address the problems of incomplete and hard to understand counterexamples, various extensions to linear counterexamples have been proposed [6, 24, 4] but quantified expressions have not been considered.

We propose a new algorithm for generating counterexamples for first order quantified temporal properties expressible in the temporal description logic ALCCTL [25], a decidable combination of the description logic ALC [2] and CTL [10]. ALCCTL has been applied for verifying properties of web documents [26] and technical manuals [21, 23]. The proposed algorithm builds upon the concept of evidence tree introduced in [27]. It extends the algorithm presented in [27] in the following two aspects:
1. Support of incremental generation of evidence trees. A coarse counterexample is provided initially which can be refined step-by-step to the desired level of detail. This prevents overwhelming the user with details of complex error scenarios and increases the responsiveness of the system.

2. Support of user interaction. The user may choose where and in which way a given counterexample is refined. This supports the successive exploration of different error scenarios according to the user’s interest, as opposed to existing model checkers which provide just a single, arbitrarily chosen counterexample.

In addition, we extend the case study in [27] as follows: 1) we demonstrate that the response time of the algorithm is sufficiently low to provide smooth user interaction even for web documents with several thousands of web pages; 2) we summarize the results of a new study on the scalability of the approach for documents up to 4000 pages.

The rest of the paper is organized as follows: first, the research issues of this paper are presented. After that, some technical preliminaries on ALC, CTL and model checking are summarized, followed by the description of the proposed structures and algorithms for counterexample generation. In the sequel, experimental results are presented before discussing related work and concluding the paper with a brief summary and outlook.

2 Issues on Counterexample Generation

Example 1 (Linear Counterexample)

Figure 1: Web document on data structures (dotted arrows indicate omitted pages)

As an example, let us consider a web-based learning document. It deals with basic data structures such as binary trees and heaps that are presented in terms of a web document hypertext (Figure 1). For simplicity, we assume that the web pages of the document form the finite set of states $S$ and hyperlinks between web pages form the set of transitions $T \subseteq S \times S$ of the Kripke structure $M$ to be checked.

The document contains formal definitions, explanations, illustrations, and examples of the basic terminology. For instance in Figure 1), page $p_{20}$ contains definitions of the terms “Tree”, “Binary Tree”, and “Heap”, as well as the explanations for “Tree” and “Binary Tree”. After that, the reader can either see an illustration of “Tree” on page 21 or an example of “Binary Tree” on page 22 ($p_{21}$ and $p_{22}$ in Figure 1). Let us assume that each defined term needs to be explained on the same page and illustrated by a pertinent example on one of the next pages. This property can be represented in ALC as

$$AG(defined \sqsubseteq explained \sqcap EX \text{ exemplified})$$

which is equivalent to the quantified CTL formula

$$AG(\forall t \in Term : defined(t) \rightarrow explained(t) \land EX \text{ exemplified}(t))$$
“On all paths it generally holds (AG) for each term $t$ ($\forall t \in \text{Term}$) that if $t$ is defined ($\text{defined}(t)$) then it is explained ($\rightarrow \text{explained}(t)$) and it is exemplified on a certain next page ($\land \text{EX}\text{exemplified}(t)$)”. 

If the set $\text{Term}$ in formula (2) is finite, the formula can be reduced to propositional CTL [10] and verified by model checkers such as NuSMV [5]. Actually, the property is not satisfied in Figure 1 because, for instance, there is no explanation for “Heap” defined on page $p20$. Counterexamples provided by the current model checkers contain a trace $(p1, p2, ..., p20)$ from the initial page $p1$ to page $p20$. $\square$

Although such a trace often becomes long, it does not give much information on why the property is violated. The following questions arise:

Q1) Where on a counterexample trace is the property violated? As for Example 1, the property is violated on page $p20$, but also pages $p21$ and $p22$ may be involved in certain error scenarios.

Q2) Which objects violate the property? As for Example 1, the property is violated for terms “Tree” and “Heap” but not for term “Binary Tree”.

Q3) Why is the property violated? As for Example 1, more than one reason can be suggested. The property is violated for term “Tree” on page $p20$, because none of the next pages $p21$ and $p22$ contains an example of “Tree”. The property is violated by term “Heap”, because no explanation is given for “Heap” on page $p20$ or, alternatively, because none of the next pages $p21$ and $p22$ contains an example of “Heap”.

Questions Q1) and Q2) refer to two dimensions of error localization. For answering Q1), the states in the state transition system $M$, which are involved in a specification violation, are determined. Q2) corresponds to bindings of the quantified variable $t$ in the formula (2) that invalidate the formula w.r.t. a given model $M$. In previous approaches on counterexamples, quantified variables have not been considered which leaves this important dimension of error localization unexploited. To answer Q2), we extend counterexamples towards subset expressions of $\text{ALCCTL}$.

Question Q3) refers to explaining why a property is violated. We observe, that even in simple scenarios the cause of a property violation can be diverse and complex. Complete explanations of property violations may be time consuming to generate and may overwhelm the user with detail. To address these issues, we 1) provide differently detailed views on an error scenario by structuring counterexamples as trees, and 2) support the incremental refinement of counterexample trees down to the desired level of detail in interaction with the user.

3 The Temporal Description Logic $\text{ALCCTL}$

$$
p, q \rightarrow C \subseteq D \mid \neg p \mid p \land q \mid \text{EX} p \mid \text{AF} p \mid \text{E}(p \lor q)
C, D \rightarrow A \mid \neg C \mid C \cap D \mid \exists R.C \mid \text{EX} C \mid \text{AF} C \mid \text{E}(C \cup D)
$$

Table 1: $\text{ALCCTL}$ syntax definition

Table 1 shows the syntax definition of a base of $\text{ALCCTL}$ connectives over the symbols $\mathcal{C} \cup \mathcal{R}$ where $\mathcal{C}$ is a set of unary predicates (atomic concepts) representing sets and $\mathcal{R}$ is a set of binary predicates (atomic roles) representing relations. In Formula (1), defined, explained, and exemplified are atomic concepts. Basic $\text{ALCCTL}$ formulae are of type $C \subseteq D$ ($C$ is a subset of $D$) where $C$ and $D$ are concept expressions. According to the second row of Table 1, concepts can be formed by $\text{ALC}$ connectives such
as A (A ∈ ℂ atomic concept), ¬C (complement), C ∩ D (intersection), and ∃R.C (quantified role R ∈ ℌ). In addition, CTL temporal connectives can be applied to form “temporal concepts”: EX C represents the set of objects which are elements of C in some next state; AF C represents the set of objects which are on all paths eventually elements of C; E(C U D) represents the set of objects which are on some path element of C until they are element of D. In Formula (1), explained ∩ EX exemplified is a (non-atomic) concept.

Any of the usual Boolean, ALC, or CTL connectives such as p ∨ q (disjunction), AG p (all paths generally p), C ∪ D (union), or ∀R.C (universal quantification on roles), can be expressed in the connectives of Table 1. The CTL fragment of ALCCTL is ALCCTL without concept constructors (second row of Table 1) and with concept subsumption C ⊆ D being replaced by atomic propositions. In this way, ALCCTL subsumes CTL.

The semantics of ALCCTL is defined w.r.t. structures M = (S, T, Δ, I) where S is a set of states, T ⊆ S × S is a left-total transition relation, Δ is a set of objects of interest called interpretation domain, and I is a state-dependent interpretation of atomic concepts and roles in such a way that for each s ∈ S, A ∈ ℂ, and R ∈ ℌ it holds: A^(I(s)) ⊆ Δ and R^(I(s)) ⊆ Δ × Δ. In this paper, we assume both S and Δ to be finite and non-empty.

Example 2 (ALCCTL Structure)
As an example of an ALCCTL structure, consider M = (S, T, Δ, I) were

\[ S = \{s0, s1, s2\} \]
\[ T = \{(s0, s1), (s0, s2), (s1, s2), (s2, s1)\} \]
\[ Δ = \{tree, heap\} \]
\[ Task^{I(s0)} = \{heap\} \] there is a task on the topic heap in s0
\[ Solution^{I(s1)} = \{heap\} \] there is a solution on heap in s1
\[ Test^{I(s2)} = \{tree, heap\} \] there is a test on tree and heap in s2

Figure 2 depicts the state transition graph (S, T). The states are annotated with non-empty interpretations of concepts.

![Figure 2: ALCCTL structure](image)

The semantics of ALCCTL defines when a formula f holds in M at a state s, denoted as M, s ⊨ f or s ⊨ f if M is understood. It extends the interpretation I to non-atomic concepts. For instance,\n\[-C|^I(s) = Δ \setminus C|^I(s), \quad (C \cap D)^|^I(s) = C|^I(s) \cap D|^I(s), \quad \text{and} \quad (EX \ C)^|^I(s) = \bigcup_{s' \in T(s)} C|^I(s') \] where T(s) denotes the T-image \( \{s' \in S \mid (s, s') \in T\} \) of s. Further,

\[ s \models C \subseteq D \text{ iff } C|^I(s) \subseteq D|^I(s) \]
\[ s \models \neg p \text{ iff } s \not\models p \]
\[ s \models p \land q \text{ iff } s \models p \text{ and } s \models q \]
\[ s \models EX \ p \text{ iff } \exists s' \in T(s) : s' \models p \]

s ⊨ AF p iff in each infinite path \((s_0, s_1, \ldots)\) in \((S, T)\) starting from \(s\) there is a state \(s_i\) such that \(s_i \models p\). s ⊨ E(p U q) iff there is such a path \((s_0, s_1, \ldots, s_n)\) in \((S, T)\) starting from \(s\) that \(s_n \models q\) and for each \(i \in \{0..n-1\} : s_i \models p\).
In this paper, we discuss counterexamples for $ALC\text{CTL}^+_R$ which is $ALC\text{CTL}$ without quantified roles $\exists R.C$ but extended with concept assertions $C(a)$ where $C$ is a concept and $a \in \Delta$ is a domain object. The semantics of concept assertions is $s \models C(a)$ iff $a \in C^s$. We disregard quantified roles in this paper merely because of space limitations. The algorithm presented in this paper can be extended to handle quantified roles by integrating the respective parts of the algorithm in [27].

**Example 3 (ALCCTL Semantics)**

Let $M = (S, T, \Delta, I)$ as in Example 2. Then

- $s_0 \not\models Solution(heap)$ because $heap \not\in Solution^{I(s_0)}$
- $s_1 \models Solution(heap)$ because $heap \in Solution^{I(s_1)}$
- $(EX\ Solution)^{I(s_0)} = \{heap\}$ because $T(s_0) = \{s_1, s_2\}$, $Solution^{I(s_1)} = \{heap\}$, and $Solution^{I(s_2)} = \emptyset$
- $(EX\ Solution)^{I(s_1)} = \emptyset$ because $T(s_1) = \{s_2\}$ and $Solution^{I(s_2)} = \emptyset$
- $s_0 \models Task \sqsubseteq EX\ Solution$ because $Task^{I(s_0)} \subseteq (EX\ Solution)^{I(s_0)}$
- $s_1 \models Task \sqsubseteq EX\ Solution$ because $Task^{I(s_1)} \subseteq (EX\ Solution)^{I(s_1)}$

\[\square\]

**Definition 4 (Model Checking Problem of ALCCTL)**

The model checking problem of ALCCTL is to decide if $M, s \models f$ for a given structure $M = (S, T, \Delta, I)$, state $s \in S$, and ALCCTL formula $f$.

A detailed description of the syntax and semantics as well as a polynomial model checking algorithm for ALCCTL is given in [25].

**Example 5 (Model Checking ALCCTL)**

Consider the ALCCTL formula

$$f = E((Task \sqsubseteq EX\ Solution) \lor \neg (Test \sqsubseteq \bot))$$

“There is a path (E) with the following properties: for each exercise task (Task $\sqsubseteq$), a solution is reachable in one step (EX Solution) until (U) there is a test ($\neg (Test \sqsubseteq \bot)$).” $\bot$ is an abbreviation for the empty concept $A \sqcap \neg A$. Hence, $s \models \neg (Test \sqsubseteq \bot)$ iff $Test^s \neq \emptyset$.

Let $M = (S, T, \Delta, I)$ be the ALCCTL structure of Example 2. Then $M, s_0 \models f$. This is because there is a path $(s_0, s_1, s_2)$ in $(S, T)$ starting from $s_0$ such that $s_0 \models Task \sqsubseteq EX\ Solution$ and $s_1 \models Task \sqsubseteq EX\ Solution$ (cf. Example 3) and $s_2 \models \neg (Test \sqsubseteq \bot)$.

The model checking algorithm given in [25] calculates the interpretations of concepts and sub-formulae in $f$ as depicted in Figure 3. In step 1), the interpretation of the non-atomic concept EX Solution is calculated for each state. In steps 2) through 4), it is determined for each state $s \in S$ and sub-formula $f'$ of $f$, whether $s \models f'$. The results in bold face are used to obtain the final result of $s_0 \models f$ in step 5). The intermediate results of steps 1) through 4) provide the basis for incremental and interactive counterexample generation as proposed subsequently.

\[\square\]

4 Generating Counterexamples

Consider an ALCCTL structure $M = (S, T, \Delta, I)$, a state $s \in S$, and such an ALCCTL formula $f$ that $M, s \not\models f$. Our aim is to generate counterexamples to $M, s \models f$ that isolate (Q1) the states in $S$ and (Q2) the objects in $\Delta$ involved in some error scenario, and explain (Q3) why a given property is violated for these objects and states (cf. section 2). To avoid information overload by bulky counterexamples, we
structure counterexamples hierarchically w.r.t. the expression tree of the verified formula and build them incrementally in interaction with the user. This way, the correspondence between parts of counterexamples and parts of the violated formula is revealed and the step-by-step analysis of complex error scenarios is supported.

4.1 Representation of Evidence

A counterexample for a formula \( f \) may contain witnesses for subexpressions of \( f \). To generalize from witnesses and counterexamples, we use the term evidence (cf. [4]). A counterexample is an evidence for \( M,s \not\models f \) while a witness is an evidence for \( M,s \models f \). We model evidence as an ordered tree obtained from the propositional reduction of an ALCCTL formula \( f \) w.r.t. \( M \) and \( s \) [25].

**Example 6 (Structure of Evidence)**

Consider the ALCCTL formula \( f \) of Example 5. Let \( p = \text{Task} \subseteq \text{EX Solution} \) and \( q = \neg (\text{Test} \subseteq \bot) \), i.e., \( f = E(p \cup q) \). Assume such a structure \( M = (S,T,\Delta,I) \) and state \( s_0 \in S \) that \( s_0 \models f \).

A suitable evidence for \( s_0 \models f \) is a path \((s_0, s_1, \ldots, s_n)\) in \( (S,T) \) on which it holds: \( s_0 \models p, s_1 \models p, \ldots, s_{n-1} \models p, \) and \( s_n \models q \). Hence, the evidence for \( s \models f \) should include the evidences for \( s_i \models p \) \((i \in \{0..n-1\})\) and \( s_n \models q \) as sub-evidences. Figure 4 depicts the structure of evidence for \( s_0 \models f \).

The edges of an evidence tree can be read as “because” (cf. caption of Figure 4): they associate a state expression \( s \models f \) with a finite sequence of state expressions \((s_0 \models f_0, s_1 \models f_1, \ldots, s_n \models f_n)\) being the reason for \( s \models f \) (note that \( f_i \) may be equal to \( f_j \) for \( i \neq j \)). Recursively generating evidences for each of the obtained state expressions \( s_i \models f_i \) results in a tree that can be built top-down based on intermediate model checking results.
Example 7 (Top-Down Construction of Evidence Tree)

Let $f$ be the ALCCTL formula and $M = (S,T,\Delta,f)$ the ALCCTL structure of Example 5. Figure 5 illustrates how a branch of the evidence tree for $s0 \models f$ is built in seven iterations. In an initialization step, the root node of the evidence tree is set to the state expression for which an evidence should be provided (Figure 5 top).

Figure 5: Part of the evidence tree for $f = E(Task \sqsubseteq EX Solution \sqsubseteq (Test \sqsubseteq \perp))$ after seven iterations.

1. Iteration: $f$ is of type $E(p \sqcup q)$ where $p = Task \sqsubseteq EX Solution$ and $q = \neg (Test \sqsubseteq \perp)$. According to Example 6), the first step in providing evidence for $s0 \models E(p \sqcup q)$ is to find a sequence $(s_0 \models p, \ldots, s_{n-1} \models p, s_n \models q)$ where $(s_0, \ldots, s_n)$ is a path in $(S,T)$ starting from $s0$. By analyzing the intermediate model checking results in Figure 3, we find the path $(s0, s1, s2)$ where

$$s0 \models Task \sqsubseteq EX Solution \quad (row\ 2\ in\ Figure\ 3)$$
$$s1 \models Task \sqsubseteq EX Solution \quad (row\ 2\ in\ Figure\ 3)$$
$$s2 \models \neg (Test \sqsubseteq \perp) \quad (row\ 4\ in\ Figure\ 3)$$

The resulting sequence of state expressions is added as a child node to the root node of the evidence tree (Figure 5, 1. Iteration).

2. Iteration: Let us assume that the user is interested now why $s2 \models \neg (Test \sqsubseteq \perp)$ (Figure 5, rhs of node obtained in the 1. Iteration). By semantics of negation “$\neg$” we get: $s2 \models \neg (Test \sqsubseteq \perp)$ because $s2 \not\models Test \sqsubseteq \perp$ (Figure 5, 2. Iteration).

3. Iteration: The user may now want to know why $s2 \not\models Test \sqsubseteq \perp$ which is equivalent to $Test^{l(s2)} \not\subseteq \emptyset$. Each element of $Test^{l(s2)} = \{tree, heap\}$ (Figure 3 rhs top) provides evidence for $s2 \not\models Test \sqsubseteq \perp$. As for the general case $s \not\models C \sqsubseteq D$, the set of evidence objects is $C^{l(s)} \setminus D^{l(s)}$, and the set of evidences is $\{ s \models C(a) \land \neg D(a) \mid a \in C^{l(s)} \setminus D^{l(s)} \}$. In the given example, we get $\{ s2 \models Test(tree) \land \neg \perp(tree), s2 \models Test(heap) \land \neg \perp(heap) \}$ as the set of alternative evidences for $s2 \not\models Test \sqsubseteq \perp$. Let us assume that the user selects $s2 \models Test(test) \land \neg \perp(test)$ for further analysis. This extends the evidence tree to the level of the 3. Iteration in Figure 5.
4. Iteration: \( s_2 \models Test(test) \land \neg \bot(test) \) because \( s_2 \models Test(test) \) and \( s_2 \models \neg \bot(test) \). Hence, the latter two state expressions in combination provide evidence to \( s_2 \models Test(test) \land \neg \bot(test) \) which is represented by the pair of state expressions as depicted in Figure 5, 4. Iteration.

5. Iteration: Assume that the user requests evidence for \( s_2 \models Test(tree) \) obtained in the 4. Iteration. Since \( Test(tree) \) is an atomic expression, it holds by definition of \( M \) and no further explanation can be provided. This is represented by the terminal node “\( \top \)" in the evidence tree of Figure 5, 5. Iteration.

6. Iteration: An evidence for \( s_2 \models \neg \bot(tree) \) is provided in the same way as in the 2. Iteration.

7. Iteration: Similar to the 5. Iteration, no further explanation for \( s_2 \not\models \bot(tree) \) can be provided which is represented by the terminal node “\( \top \)".

The remaining branches for \( s_0 \models Task \sqsubseteq EX.Solution \) and \( s_1 \models Task \sqsubseteq EX.Solution \) (Figure 5, 1. Iteration) can be expanded in a similar way. The evidence tree is complete when all leaves nodes are terminal nodes \( \top \). The evidence provided by current model checkers for the given scenario consists of the path \((s_0, s_1, s_2)\), leaving most of the analysis work to the user.

Remark 8 (Interpretation of Evidence Tree)

The evidence tree of Example 7 contains the information for answering the questions Q1) through Q3) in section 2. It clarifies

(Q1) in which states which properties hold or do not hold. For instance, \( Task \sqsubseteq EX.Solution \) holds in states \( s_0 \) and \( s_1 \), and \( \neg(\lnot(\bot)) \) holds in state \( s_2 \) as indicated by the node of the 1. Iteration in Figure 5.

(Q2) for which objects a property holds or does not hold. For instance, \( Test \sqsubseteq \bot \) is violated in state \( s_2 \) by term \( tree \) as demonstrated by the evidence for \( s_2 \not\models \bot(tree) \) (Figure 5, 3. Iteration).

(Q3) why a property holds or does not hold. The cause of a property satisfaction or violation can be drilled down by successively expanding the nodes of the evidence tree until a terminal node is reached.

For interaction with users not acquainted in temporal logic, the evidence tree is translated into a structured error report which refers to application level objects (cf. [21, 22]). In the given case, states are mapped onto web pages, domain objects onto important terms used throughout the document, and \( ALC\text{CTL} \) formula onto high-level properties derived from specification patterns [17].

Remark 9 (Optimizations)

The amount of user interaction may be reduced by clustering subexpressions of the formula and automatically expanding branches of the tree in cases without choices. As for Example 7, just the second and third iteration include choices. In the second iteration, the user has to decide for which state expression of the sequence obtained in the first iteration further evidence should be provided. In the third iteration, the user has to choose an evidence for further analysis from a set of alternative options. The Iterations 1 and 4 – 7 can be completed without involving the user.

The size of the evidence tree may be reduced by making use of semantic equivalences. For instance, \( s \models Test(tree) \land \neg \bot(tree) \) could be simplified to \( s \models Test(tree) \) in Iteration 3 of Example 7 because \( \neg \bot(tree) \equiv true \). This would remove branches that contain just trivial information. On the other hand, applying (non-trivial) semantic optimizations may result in evidence trees that are hard to understand. Further research is necessary to find a practical approach to semantic optimization.
4.2 Generation of Evidence

We now generalize the approach sketched in Examples 7 to an algorithm. First, we introduce the basic structures for representing evidence trees as depicted in Figure 5.

\[
\text{StateExpression} = S \times \{\models, \not\models\} \times \text{ALCCTL}_{\mathbb{R}}^+ \tag{3}
\]

\[
\text{Node} \subseteq \{\top\} \cup (\text{StateExpression} \times \text{ChildNode}) \tag{4}
\]

\[
\text{ChildNode} \subseteq \bigcup_{n \in \mathbb{N}} \text{Node}^n \tag{5}
\]

A state expression (Equation (3)) is a triple \((s, v, f)\) where \(s \in S\) is a state, \(v \in \{\models, \not\models\}\) a validity indicator, and \(f\) an ALCCTL\(_{\mathbb{R}}^+\) formula. A node in the evidence tree (Equation (4)) is either a terminal node \(\top\) or a state expression \(e\) which has a finite sequence of nodes as child node (Equation (5)) that provides evidence to \(e\). We use the following abbreviations:

- State expressions \((s, \models, f)\) and \((s, \not\models, f)\) are denoted as \(s \models f\) and \(s \not\models f\), respectively.
- \(\varepsilon\) denotes the empty sequence. \((s \models f)\) and \((s \not\models f)\) denote evidence nodes \((s \models f, \varepsilon)\) and \((s \not\models f, \varepsilon)\) without a child node. For instance, \((s_2 \not\models \bot (\text{tree})))\) denotes the node obtained in the 6. Iteration of Figure 5. \((s_2 \not\models \bot (\text{tree}), \top)\) denotes the same node after the 7. Iteration.
- Let \(n\) be a node and \(s = (n_0, \ldots, n_k) = (n_i)_{i \in [0..k]}\) a sequence of nodes. Then \(s \circ n\) denotes the sequence \((n_0, \ldots, n_k, n)\) obtained by appending node \(n\) to \(s\).
- Let node = \((e, c)\) be a non-terminal evidence node. Then node.expr denotes the state expression \(e\) and node.child denotes the child node \(c\) of node.

The subsequent algorithm for generating evidence consists of three parts:

1. The main function GETEVIDENCE\((s, f)\) returns an evidence for \(M, s \models f\) or \(M, s \not\models f\), respectively. It calls the model checking algorithm CHECK as defined in [25] to determine whether \(M, s \models f\). After that, INTERACTIVEEXPAND is called to incrementally generate the evidence tree for \(s\) and \(f\) on the lines of Example 7.

2. INTERACTIVEEXPAND\((EvTree)\) expands user selected branches of an evidence tree \(EvTree\). It calls GETEVSET to calculate the set of possible child nodes of a chosen node in \(EvTree\).

3. GETEVSET\((expr)\) returns the set of options for the child node of a given state expression \(expr\).

Algorithm 10 (Evidence Generation)

In the subsequent algorithm, the structure \(M = (S, T, \Delta, I)\) is assumed to be available as a global variable.

```plaintext
function GETEVIDENCE(s, f)
    if CHECK(M, s \models f) then return INTERACTIVEEXPAND(⟨s \models f⟩);
    else return INTERACTIVEEXPAND(⟨s \not\models f⟩);
end function

function INTERACTIVEEXPAND(EvTree)
    node ← USERSELECTNODE(EvTree);
    while node \neq \top do
        EvSet ← GETEVSET(node.expr);
        if |EvSet| > 1 then node.child ← USERSELECTELEM(EvSet);
    end while
end function
```
User interaction is involved in the following functions:

- **USERSELECTNODE**(EvTree), called in lines 7 and 12, returns the node of the evidence tree selected by the user to be expanded, e.g. \( \langle s2 \models \neg(\text{Test} \subseteq \bot) \rangle \) in the 2. Iteration of Example 7.

- **USERSELECTELEM**(EvSet), called in line 10, returns the child node selected by the user from a set of options, e.g. \( \langle s2 \models \text{Test}(\text{tree}) \land \neg\bot(\text{tree}) \rangle \) in the 3. Iteration of Example 7.

Function **GETEVSET** calculates the set of evidences for a state expression \( expr = (s, v, f) \) by matching it against a list of possible cases. In **GETEVSET**, \( s \in S \) is a state, \( C, D \) are \( \text{ALCCTL}^+_a \) concepts, \( A \) is an atomic concept, \( A \in \Delta \) is a domain object, and \( p, q \) are \( \text{ALCCTL}^+_a \) formulae.

If the parameter \( expr \) is a state expression of type \( s \models A(a) \) or \( s \not\models A(a) \) (line 19), a terminal evidence \( \top \) is returned (cf. 5. and 7. Iteration in Example 7). In the case of other concept assertions \( \neg C(a) \), \( C \land D)(a) \), etc., **GETEVSET** is called with the reduced expression. For instance, \( \text{reduce}(s \models \neg(C(a))) \) returns \( s \models \neg(C(a)) \), \( \text{reduce}(s \models (C \land D)(a)) \) yields \( s \models C(a) \land D(a) \), and \( \text{reduce}(s \models E(C \cup D)(a)) \) results in \( s \models E(C(a) \cup D(a)) \).

As for the other connectives, witnesses (\( \models \)) and counterexamples (\( \not\models \)) are distinguished. In line 21, a witness for \( s \models C \subseteq D \) is generated. Such a witness demonstrates that for each \( a \in \Delta \): \( s \models \neg C(a) \) or \( s \models D(a) \) which is equivalent to \( s \not\models C(a) \land \neg D(a) \). Hence, the tuple \( (\langle s \not\models C(a) \land \neg D(a) \rangle)_{a \in \Delta} \) is returned as an evidence for \( s \models C \subseteq D \) in line 21.

In line 22, a counterexample for \( s \models C \subseteq D \) is generated along the lines of Iteration 3 in Example 7. Line 23 corresponds to Iterations 2 and 6 in Example 7. The nodes obtained in Iterations 1 and 4
of Example 7 correspond to the cases of lines 31 and 25, respectively. \textbf{FINDPATH}(M, s, p, q) in line 31 applies breadth-first-search to find the set of shortest paths \((s_i)_{i \in \{0..n\}}\) in \(M\) starting from \(s\) such that \(s_0 = p, ..., s_{n-1} = p\) and \(s_n = q\). Such sequences are witnesses for \(s \models E(p \cup q)\). Similarly, \textbf{FINDLOOP}(M, s, p) in line 30 searches for shortest paths \((s_i)_{i \in \{0..n\}}\) in \(M\) from \(s\) such that \(s_n = s_j\) for some \(j \in \{0..n - 1\}\) (loop property) and \(s_i \models p\) for each \(i \in \{0..n\}\). Such sequences are counterexamples for \(s \models AF\ p\).

Note that witnesses for \(s \models AF\ p\) would have to demonstrate that on all paths starting from \(s\) eventually \(p\) holds. Since, in general, there are infinitely many such paths, a compact evidence for \(s \models AF\ p\) cannot be provided. The same holds in the case of \(s \not\models E(p \cup q)\). As a consequence, a terminal evidence \(\top\) is returned in lines 29 and 32.

\textbf{Remark 11 (Soundness, Completeness, and Termination of the Algorithm)}

A preliminary, non-incremental version of Algorithm 10 has been proven to be \textit{sound} for \(ALC\) and \textit{complete} for a larger fragment of \(ALC\) [27] than previous counterexample algorithms [18, 7, 6]. Function \textbf{GETEVSET} terminates because each calculated set is finite. Function \textbf{INTERACTIVEEXPAND} is terminated by the user. \(\square\)

5 Experimental Results

The proposed algorithm has been implemented in Java and integrated in our \(ALC\) model checker for document verification [26]. The runtime results have been acquired on a notebook computer with Intel Core 2 Duo processor at 2.93 GHz, 4 GB RAM, 64 GB SSD, running Windows 7 (32 Bit) and Java 6 update 19. The proposed algorithm has been compared with the CTL model checker NuSMV 2.4.3 [5].

5.1 Evaluation Case

As an evaluation case, we used an XML-based training document on industrial robots which is implemented in the SCORM [1] standard for web-based e-learning content. The document consists of 90 web pages, 79 of them being represented as “states” in the \(ALC\) model \(M\) that is generated from the XML markup by a software component. The document has been checked against 25 criteria each represented both as an \(ALC\) formula and as a CTL formula for comparison with NuSMV. 5 formulae were found violated. For each satisfied and violated formula, a complete evidence tree has been calculated using Algorithm 10. User interactions were simulated by a depth-first expansion of nodes in lines 7 and 12, and a random selection in the case of alternatives in line 10 of Algorithm 10.

5.2 Results

The rows in the center of Table 2 summarize the sizes of generated evidence as compared to counterexamples generated by NuSMV. Since the document structure \(M\) and the verified properties have been chosen in such a way that they can be represented equally well in \(ALC\) and CTL, the counterexamples of \(ALC\) and of NuSMV are quite similar in size and range between 1 and 80 states in the case of NuSMV, and between 3 and 85 nodes in the case of \(ALC\). However, the \(ALC\) counterexamples are structured as trees, breaking down large error scenarios into comprehensible units. The largest evidence tree (height 17, 85 nodes) was constructed for a formula with as many as 42 subexpressions. It is almost impossible to manually analyze the linear counterexamples returned by NuSMV for such complex properties. The evidence sets for each node contained between 1 and 9 elements, i.e., the user could choose one of at most nine options in line 10 of Algorithm 10 to expand the current node.

The counterexamples provided by both \(ALC\) and CTL lead to 6 “error states”, i.e., states that correspond to defective web pages. However, while in the case of \(ALC\) these states are clearly identified in the evidence tree, it requires a considerable amount of manual effort to find the relevant states in
### Table 2: Size and results of the case study on XML documents

<table>
<thead>
<tr>
<th></th>
<th>ALCCTL</th>
<th>CTL (NuSMV)</th>
</tr>
</thead>
<tbody>
<tr>
<td># web pages / states</td>
<td>90 / 79</td>
<td></td>
</tr>
<tr>
<td># violated formulae</td>
<td>5 of 25</td>
<td></td>
</tr>
<tr>
<td>evidence trees for satisfied formulae</td>
<td></td>
<td></td>
</tr>
<tr>
<td>height</td>
<td>2 – 7</td>
<td>– (no witnesses)</td>
</tr>
<tr>
<td># nodes</td>
<td>2 – 146</td>
<td>–</td>
</tr>
<tr>
<td># elements of evidence set per node</td>
<td>1 – 8</td>
<td>–</td>
</tr>
<tr>
<td>evidence trees for violated formulae</td>
<td></td>
<td></td>
</tr>
<tr>
<td>height</td>
<td>3 – 17</td>
<td>1 (linear counterexamples)</td>
</tr>
<tr>
<td># nodes (CTL: # states)</td>
<td>3 – 85</td>
<td>1 – 80</td>
</tr>
<tr>
<td># elements of evidence set per node</td>
<td>1 – 9</td>
<td>– (no alternatives provided)</td>
</tr>
<tr>
<td># error locations (states) found</td>
<td>6</td>
<td>6 (required manual analysis)</td>
</tr>
<tr>
<td># error objects found</td>
<td>5</td>
<td>–</td>
</tr>
<tr>
<td>total runtime</td>
<td>280 ms</td>
<td>660 ms</td>
</tr>
<tr>
<td>runtime of model checking</td>
<td>31 ms</td>
<td>30 ms</td>
</tr>
<tr>
<td>runtime of evidence generation</td>
<td>30 ms</td>
<td>310 ms</td>
</tr>
<tr>
<td>rest (doc. analysis, model generation)</td>
<td>219 ms</td>
<td>320 ms</td>
</tr>
</tbody>
</table>

counterexample sequences returned by NuSMV. In addition to error locations, the ALCCTL counterexamples identified a total of 5 “error objects”. Error objects are bindings of quantified variables in violated subexpressions (cf. Example 1 and Q2 in section 2). In the given case, they represent incorrect properties of parts of web pages. For instance, a “test solution”, which has been tagged as an “information unit” by mistake, has been detected as an error object but has not been reported in the counterexamples provided by NuSMV.

The lower part of Table 2 summarizes the runtime results of the experiment. Although providing more structured and accurate evidence, the proposed algorithm performed better than NuSMV. We assume that it is easier to extract evidence in the case of an explicit representation of the state space as applied in ALCCTL model checking than in the case of a symbolic representation used by NuSMV.

#### 5.3 Performance

For determining the scaling of runtime in the document size, a series of 8 documents consisting of 16 through 128 chapters of 32 pages each has been synthesized. Each of these documents have been checked against 10 formulae. Seven of them were satisfied and three of them were violated. A complete evidence tree for each satisfied and violated formula was generated by simulating user choices as described in section 5.1.

Table 3 shows the results for four cases of the experiment. The rows in the center of Table 3 report on the size of the largest evidence tree for each document. The height of an evidence tree merely depends on the corresponding formula. Since the same set of formulae were checked on each document, the heights of the resulting evidence trees do not vary across different documents. In contrast, the number of nodes of the largest evidence tree and the size of the largest evidence set per node grow proportionally in the size of the document. When evidence sets grow beyond 50 elements, the interactive exploration of each case becomes infeasible. The height of evidence trees grows linearly in the size of the formula and thus may become large for very complex properties. Promising strategies to reduce the height and width of evidence trees are: 1) clustering larger parts of formulae into macro operators, based on specification
### Table 3: Results on larger documents

<table>
<thead>
<tr>
<th># web pages / states</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td># violated formulae</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>size of largest evidence tree</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>height</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td># nodes</td>
<td>900</td>
<td>1796</td>
<td>3588</td>
<td>7172</td>
</tr>
<tr>
<td># elements of the largest evidence set</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
</tr>
</tbody>
</table>

| total runtime               | 0.7 s | 1.3 s | 2.1 s | 3.6 s |
| runtime of model checking   | 31 ms  | 78 ms  | 140 ms | 280 ms |
| runtime of evidence generation | 16 ms  | 32 ms  | 47 ms  | 172 ms |
| worst case user interaction response time | <10 ms | 15 ms  | 32 ms  | 93 ms  |
| rest (doc. analysis, model generation) | 0.65 s | 1.2 s  | 1.9 s  | 3.1 s  |

The rows on the bottom of Table 3 summarize the runtime results. Even for large documents, the total runtime remains below 5 seconds. The runtime is dominated by the time for document analysis and model generation and scales approximately linearly in the document size. Evidence generation takes less than 5% of the total runtime. Most important for smooth user interaction is the response time of the system when expanding the evidence tree. The response time is dominated by the runtime for calculating the evidence set for a given state expression (line 9 in Algorithm 10). Even in the case of very large documents, the worst case response time remains below 100 ms, sufficiently low for smooth user interaction.

### 6 Related Work

[18] describes the basic method of generating linear counterexamples for CTL which is still adopted in state-of-the-art model checkers such as NuSMV [5] or SAL [8]. [7] suggests a method for generating richer “tree-like” counterexamples. The tree structure of the counterexamples corresponds with computation trees of the verified model. We structure counterexamples along the expression tree of the verified formula. This clarifies the correspondence between parts of the counterexample and parts of the violated formula and supports tracking the cause of a property violation down to the desired level of detail. Further, a higher level of completeness and detail is obtained than in [7] because we also consider Boolean connectives, the subset operator of $ALC$, and both witnesses and counterexamples for EX.

There have been a number of efforts for addressing the problem of bulky counterexamples. [16] suggests efficient algorithms, based on transitions shuffling, for approximating the smallest counterexample in on-the-fly model checking. [20] and [24] define a method for minimizing variable assignments in CTL counterexample traces which makes it simpler for the user to find areas of interest. [12, 13, 11] and [3] localize errors in C programs based on model checking results by comparing incorrect and correct runs of the program. Incremental generation of counterexample and witnesses w.r.t. the user’s interest, however, has not been considered.

[4] proposes a framework for counterexample generation and exploration, based on “proof-like” counterexamples [14]. Counterexamples and witnesses provided by a symbolic CTL model checker are annotated with proofs that explain why a property holds in a given state of a model. While these proofs support experts in analyzing counterexamples they may be difficult to understand for users not acquainted
in proof systems and proof rules.

A first method for finding counterexamples for the CTL fragment of ALCCTL has been described in our previous work in [25] and demonstrated in case studies on checking the consistency of technical documentations [21,23]. [27] proposes a formal definition and analysis of evidence trees as well as a first algorithm for generating them. This paper extends previous work towards incremental and interactive generation of evidence trees based on intermediate model checking results. In addition, the case study in [27] is extended towards larger and more complex documents and towards the analysis of the system’s response time in interactive evidence generation.

7 Conclusion

We have presented a new algorithm for the incremental generation of tree-structured counterexamples and witnesses for properties of web documents expressed in the temporal description logic ALCCTL. The algorithm supports exploring alternative error scenarios according to the user’s interest, instead of providing just a single, arbitrarily chosen counterexample as existing model checkers do. The generated counterexamples identify both the parts of the model and the parts of the formula involved in some error scenario, and support the step-by-step analysis of the cause of a property violation. The runtime performance of the algorithm scales up to application-relevant problem sizes. The worst case response time remains below 100 ms even for documents with several thousands of web pages. The presented approach thus provides a solid basis for generating structured, precise, and user adaptable error reports. Issues of future work include the optimization of the algorithm to minimize the size of evidence and the amount of user interaction, the visualization of application level reports generated from evidence trees, and the evaluation of the usefulness of the approach for differently experienced users.

8 Acknowledgements

This work is funded by the program “Research at International Science and Technology Centers” of the German Academic Exchange Service (DAAD). The case study reported in section 5 originated from the Verdikt project [26]. We thank Burkhard Freitag for continuing support.

References

Incremental Refinement of Counterexamples

Weitl and Nakajima


