Final Bachelor's Paper

Oryx: Embedding Business Process Data into the Web

Martin A. Czuchra

martin.czuchra@student.hpi.uni-potsdam.de

June 30, 2007

Contents

List of Figures

1. Introduction

This paper is part of the documentation for the Oryx Business Process Editor, developed as a bachelor's project at the Hasso Plattner Institute at the University of Potsdam. A related project implemented a back end to the editor that, in addition to storing process models, is able to instantiate and execute them. The server is named Thanis, and will be referenced by this name in the following documentation.

The aim of the Oryx project was to create a web-based editor that is able to run in at least one browser family. Since we considered standard conformance a very important aspect of a web application editing business processes, we put very much effort into making it conform to the W3C specifications. Since we want to allow other tools to interoperate with the editor, no proprietary protocols should be used for communication between the components, or between client and server.

There are lots of process modeling languages. While concentrating mainly on Business Process Modeling Notation (BPMN, [1]) support, the editor has to be easily extensible to implement a wide range of languages. Figure 1.1 shows an example process in the Thanis stencil set. We call a package containing a new language a stencil set [8]. Regarding the extensibility of the editor's functions and features, we decided to build a plugin infrastructure that makes functionality easy to extend [10].

Furthermore, the UI has to be be both appealing to business process designers and to domain specialists. This is a very challenging aim and it is even stressed by the need to design the editor similar to a traditional desktop application. This is due to the high responsiveness a traditional graphing tool provides when editing, and the resulting desktop feeling has been ported into the web with Oryx.
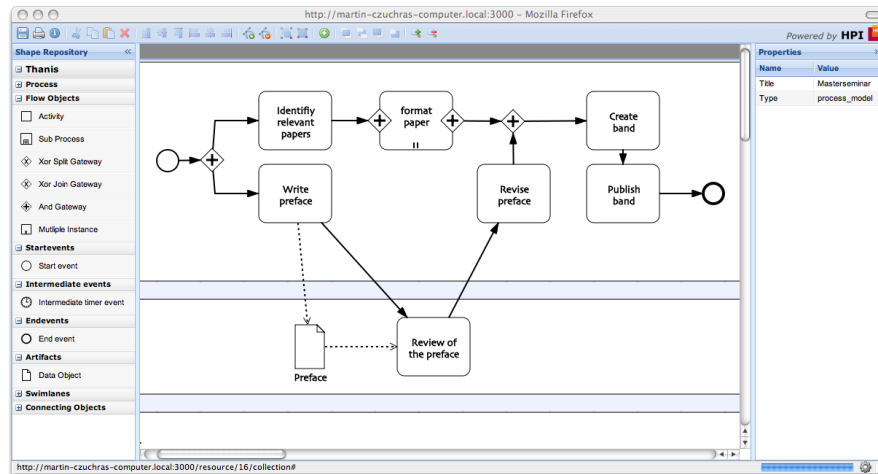
Figure 1.1: A screenshot of the Oryx editor with the Thanis stencil set and an example process.

This paper will concentrate on the documentation of Oryx' data management, which includes data storage inside the editor, interfaces with other applications and server communication. There are papers describing the core editor functionality and the plugin subsystem (see [10]), the stencil set specification (found in [8]) and the implementation of the BPMN stencil set (see [9]).

1.1 Chapter Overview

I have divided this paper into the chapters introduction, evaluation, implementation, problems, conclusion and outlook.

The introduction will provide a brief overview over the editors purpose and related work and it provides this chapter overview. The evaluation chapter will briefly cover BPMN fundamentals, discuss different data storage techniques, and give an overview over possible server communication techniques.

The chapter implementation will then cover the Oryx data management en detail. I will document the eRDF triple's internal representation, querying techniques, the eRDF parser, the triple store, the triple manipulation API, the resource management, the design of a domain specific language, and I will provide a brief discussion on data management performance. The chapter problems will concentrate on side effects and ambiguities when dealing with triples in the Oryx data management. At the end of this paper, I will additionally provide a conclusion and an outlook.

2. Evaluation

The Oryx editor's architecture defines the application in its final configuration to integrate into a page that is rendered from a business process resource located on a server into the client's browser. Such a resource is uniquely identified by a URI.

Since the editor uses SVG for its graphical representation of processes and most of the web is written in different dialects of hypertext markup, the best solution for the format of the process resource being delivered to the browser is XHTML, which, unlike HTML allows seamless inline integration of SVG into one document.

There will probably be other applications running on the same page as the Oryx editor, possibly manipulating the process model or even executing an instance. Therefore, there is a need to synchronize their interaction with the underlying on-page resource data to avoid data loss due to concurrent access to the DOM. Additionally, to permanently store data, there is a need to propagate on-page changes that have been performed back to the server the process originates from.

In order to choose the best technology for data management in Oryx, I have evaluated several ways to store data in the page's DOM and to persist it on the server. In this chapter I will discuss four different data storage approaches and why eRDF is superior to all of them in our setup.

In addition to that, different bootstrapping approaches will be described, some of which were fully implemented in the development process of Oryx.

Before the evaluation of the different technologies, I will show the need to use highly structured data types when dealing with process data. I will therefore provide a brief introduction into BPMN fundamentals. See [1] for more details.

2.1 BPMN Fundamentals

BPMN knows four different types of elements in its diagrams: flow objects, connecting objects, artifacts and swimlanes [1]. The graphical representations of BPMN model elements described in this section are combined into diagrams that then include BPMN graphs[1]. So, each of the elements to be described is embedded into a graph, and every element has a set of typed properties that can be found in [9], [1]. For more information on swimlanes, also see [9], [1].

2.1.1 Flow Objects

Flow objects include events, activities and gateways.
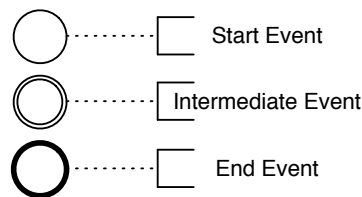
Figure 2.1: Graphical representation of different event types

Events are cut into start-, intermediate- and end events [1]. They are represented by circles with either single, double or thick stroke, as shown in figure 2.1. They may contain a symbol that determines their type. Events are triggers for or results of activities and gateways. Start events are used to introduce a new process; end events terminate running processes. Intermediate events are used for inner process communication.
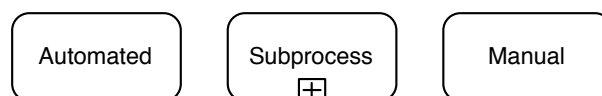
Figure 2.2: Graphical representation of different activity types

Activities represent tasks that need to be performed. They are divided into those represented by subprocesses, manual activities which need to be performed by human individuals and activities that are automatable and most likely executed by a workflow engine. Each activity is represented by a rounded rectangle as shown in figure 2.2, if necessary with symbol inscriptions at the bottom side.

---

[1] Commonly called models, not graphs. However, I want to point out the possibility of BPMN to be expressed in RDF graphs. See 2.2.2 for more details.
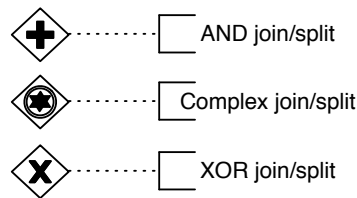
Figure 2.3: Graphical representation of some but not all gateways

Gateways describe points in the flow where decisions have to be made, splitting or joining certain threads of process execution [1]. Their visual representation is a diamond shape with a symbol for the type of the gateway inscribed into it, shown in figure 2.3. The different types are: AND-, Complex-, OR-, event based- and data based XOR joins/splits.

2.1.2 Connecting Objects
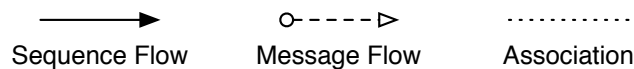
Figure 2.4: Graphical representation of flows and associations

There are three types of connecting objects in BPMN. Two of them describe flows, the message flow and the sequence flow. The other one is the association. Message flow represents message exchange between tasks belonging to different pools. Sequence flow represents the flow of execution within a thread. They provide limited chronological order on the tasks they connect. Associations are used to connect artifacts to flow objects. Figure 2.4 shows examples of all connecting objects.

2.1.3 Artifacts
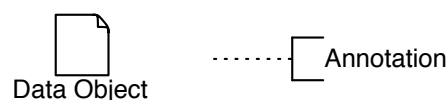
Figure 2.5: Graphical representation of data objects and annotations

BPMN divides artifacts into data objects, groups and annotations. Data objects show interaction (requiring or producing) between flow objects and data. Groups

provide differentiation on sets of activities. Annotations allow to comment processes anywhere in a diagram, thus enhancing understandability. Examples of data objects and annotations are shown in figure 2.5, groups are drawn as rounded rectangles with a dotted stroke around the activities they group.

2.2 Data storage

There are three different layers in which data is held in every Oryx setup. The topmost one is the set of instantiated shapes known to the application and the properties they have according to the stencil set [8] specification. This knowledge is accessible only to the core editor and to plugins that have been passed the appropriate facade [10]. The data is represented by memory resident Javascript objects, which share their lifetime with the shapes visible on the canvas.

The next layer of data storage is the DOM. The data stored in here has always been directly transmitted by the server and therefore should reflect the format that is favored for further communication. Manipulation of the DOM data should be restricted to the data management of the Oryx editor, as far as concerning process relevant data, to avoid unnecessary growth of synchronization needs between independent editor components.

The bottom-most layer is the persistent data storage on the server. While the data management of Oryx cannot specify the server's internal data structure, it can influence the communication protocol that is used when persisting data on the server.

In this section, I will concentrate on providing possible solutions for data storage on the middle layer, the DOM. For more information on the topmost layer, see [10]. To learn more about the possibilities in server communication, read 2.3.

2.2.1 Plain Text

The easiest way to store data in the DOM is to determine a place where to put string serializations for all objects to be stored[2]. While this is a good idea for data that is moderately structured and where there is a good chance to find a serialization readable for humans, like date or time data sets, it does not fit well in the application setup of Oryx. The data we are facing here is complex, not only moderately structured, and for many of the properties known to shapes that have to be persisted, there is a good chance not to find a human-readable serialization.

---

[2] This string serialization approach is mentioned only for the sake of completeness.

2.2.2   The Resource Description Framework

The Resource Description Framework (RDF) is a language that has been designed to describe information about resources on the web [6]. It is a W3C recommendation since February 2004. RDF already is widely deployed in the semantic web of today, and there is very good support by tools (Protegé by the Stanford University School of Medicine) and even reasoners that infer knowledge from RDF data (FaCT++ by the University of Machchester).

The basic idea behind RDF is that resources have properties which have values. It allows to define statements consisting of subject, predicate and object (RDF triples). The subject is a thing that is being described. The predicate is a certain property and the object is the value of the subject in a given property. Each object itself can then be addressed as a subject itself in additional statements [6]. Figure 2.6 shows an example of some Dublin Core metadata in RDF:

```
<?xml version="1.0" ?>
<rdf:rdf
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlsn:dc="http://purl.org/dc/elements/1.1">

    <rdf:description rdf:about="http://example.org/index.html">
        <dc:creator>Martin Czuchra</dc:creator>
    </rdf:description>

</rdf:rdf:>
```

Figure 2.6: An example RDF/XML snippet

In general, RDF can be used to build a knowledge graph, where subjects and objects are represented by nodes and predicates are displayed as edges. The result can be displayedusing an RDF graph for the triples. For some triple examples, see 3.6.

To embed RDF data directly into XHTML pages there are four possibilities, two of them using RDF/XML, the other two using embedded RDF (eRDF) or RDF/a, which will be described later on.

2.2.2.1   RDF/XML

Staying with RDF/XML, one can directly include such in a multi-namespace XHMTL document. This enables full RDF expressiveness inside web pages [3], but there are several problematic issues in this approach: The subjects that are

contained inside a page must be re-referenced inside the RDF statements each time, and there is no hint on where to put RDF data inside such a document or where to look for data on a given subject.

Avoiding the first issue, another possibility is to use RDF/XML in an additional file the web page may then reference to [3]. Using this approach, there is no need to decide on where in the XHTML markup to put new statements, as they may simply be added to the end of the existing ones. However, the need for referencing subjects in the original file remains, thus not shrinking the size of the RDF file.

The two other possibilities include using eRDF or RDF/a for this. Both these RDF sublanguages are designed to integrate well within XHTML. While eRDF is compliant with XHTML 1.0, the RDF/a specification currently has good chances to be integrated into XHTML 2.0 standard.

2.2.2.2 eRDF

Another possibility to embed a subset of RDF into XHTML documents is to use eRDF. It was originally developed by Ian Davis [5]. The speciality of eRDF is that is uses common attributes only [5], so standards conformance with XHTML is not broken. No additional elements or attributes have been added to the original XHTML standard, and CSS support is still guaranteed [5]. In addition to that, eRDF allows rudimentary CSS selection on a semantic level due to the class attribute naming scheme.

However, eRDF does not attempt to embed the full RDF model into XHTML [5]. The main drawback when comparing eRDF to the other RDF dialects is that it only supports properties that represent either URIs or literals. No support for typed properties is included, however it may be approximated as described in 3.1.1.2. The relationship between eRDF and RDF is the following: "All HTML Embeddable RDF is valid RDF, not all RDF is Embeddable RDF" [5].

One of the rules eRDF knows is the following: "meta elements in the head of a document generate a triple with: a subject equal to the document's URL, a predicate derived from the element's name attribute, a literal value equal to the content attribute" [4]. eRDF knows ten such rules that define how XHTML constructs are mapped into RDF and vice versa. To learn more about all eRDF rules, see [4]. Figure 2.7 shows an example of eRDF that produces exactly one triple according to the mentioned rule.

2.2.2.3 RDFa

RDFa is another language to embed RDF into XHTML. Like eRDF, it uses the original content of the site, adds some markup and thus adds the semantic annotation

```
<?xml version="1.0" ?>
<html xmlns="http://www.w3.org/1999/xhtml"">

    <head profile="http://purl.org/NET/erdf/profile">
        <meta name="dc-creator" content="Ian Davis" />
    </head>

    <body> [...] </body>
</html>
```

Figure 2.7: An example XHTML file with eRDF

without the need to repeat many of the data [2]. It is therefore as well suited as the other RDF dialects when it comes to expressing structured data in the form of RDF graphs. However, it extends the XHTML standard on some points by introducing meta elements in the body of an document and property attributes on elements that are unknown in the current XHTML standard.

```
<html xmlns:cal="http://www.w3.org/2002/12/cal/ical#">
    <head><title>Jo's Blog</title></head>
    <body>
    [...]
    <p class="cal:Vevent" about="#xtech_conference_talk">
        I'm giving
        <span property="cal:summary">
            a talk at the XTech Conference about web widgets
        </span>,
        on
        <span property="cal:dtstart" content="20070508T1000+0200">
            May 8th at 10am
        </span>.
    </p>
    [...]
    </body>
</html>
```

Figure 2.8: An example featuring RDFa (taken from [2])

2.2.2.4　Conclusion

Since XHTML 1.0 validity was an important requirement, we decided on usage of eRDF for internal data representation. The concept of nesting information into div elements and the possibility to use CSS selectors on semantic markup are a great gain for both Oryx and the Thanis wiki. Using a separate data file in the style of RDF/XML appeared as massive overhead, and the non-standard extensions to XHTML used in RDFa made it useless when deployed in a standards compliant environment. However, as RDFa may become the future XHTML 2.0 standard, this may result in the need to add support for this data format, too.

2.3　Server Communication

Regarding the client-server communication, there are several possibilities for how to design the interface. Common to all of them is the usage of AJAX[3]-like requests that are sent to the server and the need to stay with the communication protocol HTTP. The content of these messages, however, is subject to data management decision.

Client-server communication needs to be performed when there are changes on the client side that have to be propagated to the server, and when there are changes to the server side that have to be propagated to the client. Latter includes the changes performed by other clients working on the same model.

One of the possibilities is to use JavaScript simple object notation (JSON) as the message content for synchronization. JSON is the data format that the Oryx editor uses to define it's stencil set specifications [8]. The advantage of this approach is that JSON doesn't need to be parsed on the client side, it is sufficient to simply evaluate the JSON object as JavaScript. The main disadvantage is that the Thanis server, which is the default back end at the time, has no knowledge of JSON and would have to transform the data back to XML to store it.

Another possibility is to use XML in the exchange of messages. XHTML could be directly exchanged between client and server. While the JSON approach is focusing more on the transfer of JavaScript logic, this approach is more data-centric. The advantage is that XHTML data returned from the server can be put into the DOM without further data processing. The main disadvantage is the need to serialize the local DOM into a string when sending parts of it to the server.

Using any proprietary data structure in the client-server communication has not been considered, as it was an important requirement to use open standards in both storage and communication channels.

---

[3]Asynchronous JavaScript and XML

The final decision fell on the XML message exchange. Since the server has a deep understanding of how to handle XML, it is easy to render XML responses for client requests. The gain in performance for not having to synthesize DOM elements from JSON objects overweighed the overhead for serializing DOM segments for transfer to the server. To learn about how the client-server communication was implemented, see 3.5.

3. Implementation

3.1 Data Access

When discussing the data management of Oryx, there are a lot of implementation details that are important to both continuing the development of the editor and being able to access Oryx data from other applications running on the same page. While architectural and implementation details for the different modules of data management will be discussed later in this chapter and independently for each module, in this section, I will concentrate on giving an overview of the RDF triple as it is used in data management and the triple querying techniques available in Oryx. This section should be an entry-point for every developer interested in accessing Oryx' process data.

3.1.1 The RDF Triple

Each RDF Triple consists of a subject, a predicate and an object. However, when using eRDF to store triples, there are certain constraints on the data types of both the subject and the object. The subject is always a resource, the object can be either resource or literal. Figure 3.1 shows an example of eRDF-annotated XHTML that results in the triples displayed in figure 3.2. Resources are by convention enclosed into angle brackets and literals into quotation marks. As you can see, only the first triple of figure 3.2 has a resource object, but all share the same resource subject. The two other triples both use literal objects. []

3.1.1.1 Internal Triple Representation

With respect to the triple characteristics described in 3.1.1, the internal data structure representing a triple was chosen to be a JavaScript object with at least the

```
<div id='oryx-activity1234'>

    <a href='./edit' rel='oryx-editurl' />

    <span class='oryx-name'>Parse eRDF</span>
    <span class='oryx-bounds'>20,20,120,80</span>
</div>
```

Figure 3.1: A simple eRDF sample.

```
<#oryx-activity1234> oryx:editurl
    <http://b3mn.org/resource/activity1234/edit>
<#oryx-activity1234> oryx:name 'Parse eRDF'
<#oryx-activity1234> oryx:bounds '20,20,120,80'
```

Figure 3.2: The triples produced by figure 3.1.

following fields: subject, predicate and object. Each subject and object have to know both a type and a value. The type may only be one of the following constants: `ERDF.RESOURCE` or `ERDF.LITERAL`. The value is always a JavaScript string that in case of a resource subject or object contains a URL, in every other case the plain literal string. The predicate, however, knows no type or value, but a prefix and name.

```
var triple = {
    subject: {type: ERDF.RESOURCE, value: '#oryx-activity1234'},
    predicate: {prefix: 'oryx', name: 'name'},
    object: {type: ERDF.LITERAL, value: 'Parse eRDF'}
};
```

Figure 3.3: The notation of a triple using JSON.

Figure 3.3 shows a simple example of a triple[1] created by usage of JSON, figure 3.4 shows the generation of an `rdf:type` triple as performed by the eRDF parser (see rule six in 6). When code readability is preferred over additional helper methods, use the JSON way to write triples, otherwise, and if you wish to further process the data, use the ERDF.Triple, ERDF.Resource and ERDF.Literal constructors instead.

---

[1] This particular triple is the second one from figure 3.1.

```
var triple = new ERDF.Triple(
    (subjectType == ERDF.RESOURCE) ?
        new ERDF.Resource(subject) :
        new ERDF.Literal(subject),
    {prefix: 'rdf', name: 'type'},
    new ERDF.Resource(schema.namespace+property)
);
```

Figure 3.4: The generation of an `rdf:type` triple as performed by the eRDF parser.

3.1.1.2 Complex Data Types

When describing object values that are not resources and also are not literals, such as numbers, or more complex data types like dates, one has to use literals in eRDF triple generation and accompany every such object with the following:

- A resource URL, describing the location of the object value in order to make it possible to make new assertions on it.

- A second triple whose subject is the resource URL of the object value, whose predicate is `rdf:type`, and a resource object referencing the appropriate data type in the XSD [7] namespace (mostly by providing an XSD data type).

Changing the previous example to reflect the fact that the last triple has a complex data type object, namely `oryx:bounds-datatype`, would result in what is to see in figure 3.5. The triples generated by this XHTML snippet can be seen in figure 3.6.

This is the general approach when dealing with complex types in eRDF. However, to simplify triple manipulation and to avoid overhead in both the Oryx editor and the back end, the Oryx data management does not provide any more support for complex types other than the possibility to query for their name. Internally, all triples are stored with literal or resource objects. Whether any querying application respects the provided type information is in its responsibility only and fully optional. In fact, the Oryx editor expects all triple objects representing shape properties as described by the stencil set to be literals [10].

3.1.2 Querying Technique

There is a very simple way to access all triples from the Oryx data management. The editor knows a public object named DataManager, that itself knows a method

```
<div id='oryx-activity1234'>

<a href='./edit' rel='oryx-editurl' />

    <span class='oryx-name'>Parse eRDF</span>
    <span class='oryx-bounds -oryx-bounds-datatype'
        id='bounds345'>20,20,120,80</span>
</div>
```

Figure 3.5: An advanced eRDF sample.

```
<#oryx-activity1234> oryx:editurl
    <http://b3mn.org/resource/activity1234/edit>
<#oryx-activity1234> oryx:name 'Parse eRDF'
<#oryx-activity1234> oryx:bounds <#bounds345>
<#bounds345> rdf:type 'oryx:bounds-datatype'
```

Figure 3.6: The triples produced by figure 3.5.

named query(). The actions performed internally cause the data manager to access the triple store to find all matching triples. To learn more about the triple store, see 3.3.

The query method as seen in 3.7 accepts three parameters. The first one is supposed to be a triple's subject, the second one a triple's predicate and the last one a triple's object, all of them in the style of a general triple as described in 3.1.1.1. Any of the parameters may be null or undefined. The query method identifies all triples known to the data manager which match the following rules:

1. If a subject has been passed to the query method, this subject must equal the triple's subject in both the subject type and value.

2. If a predicate has been passed to the query method, the prefix and name of the passed predicate must either be null or undefined, or they must equal the prefix and name of the triple's predicate, respectively[2].

3. If an object has been passed to the query method, this object must equal the triple's object in both the object type and value.

---

[2] This rule is slightly different from the others, since it allows to query for all triples with a certain prefix, without specifying a name. Subjects and objects cannot be queried specifying a type only.

```
query: function(subject, predicate, object) {

    /*
     * Typical triple.
     * {value: subject, type: subjectType},
     * {prefix: schema.prefix, name: property},
     * {value: object, type: objectType});
     */

    return DataManager._triples.select(function(triple) {

        var select = ((subject) ?
            (triple.subject.type == subject.type) &&
            (triple.subject.value == subject.value) : true);

        if(predicate) {
            select = select && ((predicate.prefix) ?
                (triple.predicate.prefix == predicate.prefix) : true);
            select = select && ((predicate.name) ?
                (triple.predicate.name == predicate.name) : true);
        }

        select = select && ((object) ?
            (triple.object.type == object.type) &&
            (triple.object.value == object.value) : true);
        return select;
    });
}
```

Figure 3.7: The query method of Oryx data management.

All triples matching these rules will be collected and returned in form of an array. Such a triple array might then be traversed using the each-method to perform an action on each data set.

Figure 3.8 shows the Oryx editor's initialization of all canvases. Therefore, the editor is querying all triples that have a resource object identifying the URL `http://oryx-editor.org/canvas`. Since the subject and the predicate are passed as undefined to the query method, they may be arbitrary in the returned triples. For each triple in the result, the function passed to the each-method is then called with c being the current triple. In this function, the queried triples are being analyzed and editor instances are spawned for each triple's subject.

```
DataManager.query(
    undefined,
    undefined,
    {type: ERDF.RESOURCE, value: 'http://oryx-editor.org/canvas'}).each(

        function(c) {
            var anchor = c.subject.value;
            var id = anchor.substring(1, anchor.length);
            new ORYX.Editor(id);
        });
```

Figure 3.8: Querying all canvas resources and instantiating an editor instance for each.

Figure 3.9 queries the triples describing an `oryx:type` property for the current document. The subject of the triples to be queried is therefore a resource with an empty URL, which in every case refers to the current document. The predicate is defined to equal the desired `oryx:type` property, and the object is left undefined[3]. Ideally, this query should return exactly one triple, but since the data manager does not know about the constraints Oryx puts on certain properties, it returns an array in this case, too. In this example, the result is being stored in the `canvasType` variable for further processing. After assuring that there is exactly one such triple, one can access the object's value using `canvasType[0].object.value`.

---

[3] Consider that since this is the last parameter in the method call, the object parameter could have been omitted completely resulting in the same query semantic.

```
var canvasType = DataManager.query(
    {type: ERDF.RESOURCE, value: ''},
    {prefix: 'oryx', name: 'type'},
    undefined
);
```

Figure 3.9: Querying the oryx:type property of a canvas.

3.2 eRDF Parser

The eRDF parser of the data management module of Oryx is the heart of data retrieval; eRDF triples are being parsed from the underlying XHTML data. It is a strictly passive module that does not manipulate the DOM in any way. In this section, I will concentrate on describing the way in which the eRDF parser works and how triples are being generated.

For extensibility of the original HTML standard, a profile attribute has been added to the head element of any HTML document. This attribute is supposed to be a list of URLs defining additional meta data profiles, separated by whitespace. The profile used for eRDF is `http://purl.org/NET/erdf/profile`. Only documents containing this profile reference will be regarded containing eRDF data. All other documents will not pass the parsing routine.

eRDF itself knows the concept of schemas to identify sets of predicate prefixes used in triples. Inherently known to every RDF handling application should be the `rdf`, the `rdfs` and the `schema` schema. Knowledge about these schemas is hardcoded into the initialization routine of the eRDF parser.

To allow the recognition of both Oryx and back end data encoded in the eRDF, one has to additionally reference the following two schemas: `oryx` for the editor, and `raziel` for the back end. Every schema has to be defined in the head section of the XHTML document, using a link element that references the schema in its rel attribute and the schema's URL in the href attribute. Figure 3.10 shows the definition of a document including profile reference and all essential schemas for usage in Oryx.

A specialty of the eRDF parser implementation used in Oryx is that all schema definitions are considered triples themselves. When generating a triple, the schema definition must already be known. It is therefore recommended to define all schema references in the very top of the document's head, before any other triple generating parts of the document can be traversed.

```
<?xml version="1.0" encoding="utf-8"?>
<html xmlns="http://www.w3.org/1999/xhtml">
    <head profile="http://purl.org/NET/erdf/profile">
        [...]
        <link rel="schema.oryx" href="http://oryx-editor.org/" />
        <link rel="schema.raziel" href="http://raziel.org/" />
    </head>
    <body> [...] </body>
</html>
```

Figure 3.10: eRDF profile and schema definition for usage in Oryx

When describing the parsing process, there are two parts I would like to draw attention to: the parsing of the head of the document (3.2.1) and the parsing of the body of the document (3.2.2). Triples generated in any of these two parts are being reported to the registered triple store (see 3.3). Consider that in the process of developing a JavaScript parser for eRDF, I have merged some pairs of rules that the original specification considered distinct (compare to [4]) into one. This has been done to improve source code readability and performance (3.7).

3.2.1   Head Metadata

There are two triple production rules that have to be considered when dealing with eRDF data in the head of an XHTML document:

1. Meta elements with content and name attributes produce a triple with a subject corresponding to the document's URL, a predicate corresponding the name attribute and a literal object defined by the content attribute.

2. Link elements that have a href attribute and either a rel or a rev attribute, or both, produce triples of the following kind: A subject corresponding the document's URL, a predicate corresponding the rel attribute and an object defined by the URL in the href attribute; The possible predicates in the rev attribute produce triples with subject and object interchanged[4].

Matching elements are searched for by getting the first[5] head element in the document and then getting all immediately descending link and meta elements. First,

---

[4] This rule is a result of the merging of two distinct rules from [4].

[5] In every well-formed XHTML document, the first head element is the only one.

all link elements are being analyzed, since they may contain additional schema definitions. Then, all meta elements are checked for triple generation. All generated triples are reported to the triple store registered with the parser.

3.2.2 Body Metadata

The remaining part of the document is being parsed recursively, beginning with the document's first[6] body element. The parsing in this section of the XHTML document leaves out all elements that are not in the XHTML namespace, namely `http://www.w3.org/1999/xhtml`. This is a legitimate decision, since eRDF content is defined only inside of XHTML. Furthermore, since the Oryx editor adds lots of SVG content, and the plugins extend the DOM with elements from the Yahoo! Ext namespace [10], avoiding all non-XHTML namespaced elements dramatically improves performance (see 3.7).

The resource subject of almost every triple that is being produced in this part of parsing is either, if existent, the id attribute of the first ancestor possessing one, or, if there is no such ancestor, the document's URL. For convenience, this resource will be called the *current subject* for each element, respectively. The following rules have then to be applied on the remaining document's elements:

1. Any elements with a class attribute produce triples with the *current subject*, a predicate corresponding to the class attribute and a literal object that either corresponds to the title attribute of the element or a concatenation of all descendant text nodes, if there is no title attribute.

2. Anchor (`a`) elements[7] that have a href attribute and either a rel or a rev attribute, or both, produce triples of the following kind: A subject corresponding the *current subject*, a predicate corresponding the rel attribute and an object defined by the URL in the href attribute; the possible predicates in the rev attribute produce triples with subject and object interchanged[8].

3. Anchor (`a`) elements that produce a triple according to the previous rule, additionally generate a triple with a subject corresponding to the content of the href attribute, a predicate equaling `rdfs:label` and and a literal object that

---

[6] In analogy to the head element, in every well-formed XHTML document, the first body element is the only one.

[7] This rule is similar to to the second rule in section 3.2.1, besides the fact that head elements do not allow anchors to be set and document bodies forbid the usage of link elements.

[8] This rule is a result of the merging of two distinct rules from [4].

either corresponds to the title attribute of the element or a concatenation of all descendant text nodes, if there is no title attribute[9].

4. Image (`img`) elements that have a src and a class attribute produce triples with the *current subject*, a predicate corresponding to the class attribute and an object corresponding to the src attribute[10].

5. Image (`img`) elements that produce a triple according to the previous rule and that have an alt attribute, additionally generate a triple with a subject corresponding to the content of the src attribute, a predicate equaling `rdfs:label` and and a literal object that corresponds to the alt attribute of the element[11].

6. Every element with a class attribute that includes tokens prefixed by a hyphen produces a triple with the *current subject*, a predicate equaling `rdf:type` and an object corresponding the token in the class attribute, without hyphen. See 3.1.1.2 for more details on this.

To apply all the rules just described, every currently parsed element is being analyzed for existence of an id attribute. Every tag is then being analyzed for a class and a title attribute, and a label is being computed to be either the title attribute or the text content of that node, if no title is available.

Anchor (`a`) and image (`img`) elements undergo a special treatment according to rules 2 to 5. For anchor tags, the rel, rev, href and title attribute are being analyzed and the text content is being computed. First, all triples resulting from the rel attribute (rule 2) are being generated[12]. For each of the triples produced here, the parser then checks the need to create an additional `rdfs:label` triple according to rule 3, and generates one if the requirements are met. For this purpose, I have introduced a callback mechanism in triple generation, which in future could annotate certain triples with the creation of additional ones. In the current implementation, this functionality is used only for such `rdfs:label` triples. The triples derived from the rev attribute of the anchor element are being generated next (rule 2). When they

---

[9] This rule is a result of the merging of two distinct rules from [4].

[10] The rule summary in [4] actually describes the src attribute to become the subject. However, according to the original specification in [5], it is stated to become the object. I consider this an error in the rule summary, and apply the rule as described in this paper.

[11] The rule summary in [4] actually describes the href and title attribute here. However, according to the original specification in [5], it is stated to be the src and alt attribute. I consider this an error in the rule summary, and apply the rule as described in this paper.

[12] Actually, rule 2 describes triples to be derived from rel and rev attributes. However, only rel attributes produce `rdfs:label` triples, which is why rel and rev are processed separately here.

are all processed, the only remaining triples are those resulting from rule 6. They are being produced accordingly.

From image elements, the parser analyzes the class, src and alt attribute. From the tokens in the class attribute, triples are being produced according to rule 4. With a callback similar to the one used for rule 3, `rdfs:label` triples are being added to these ones to realize triple generation as required by rule 5.

Independently from the element type, the parser then checks for generation of general triples according to rule 1. The next step in parsing is setting the *current subject* to the id attribute of the current element, if there is one. The next parsing step is to consider all `rdf:type` triples according to rule 6. This last step is to make sure that both statements about the last subject can be made trough triples derived from class attributes in the current element and about the element itself by specifying an id and an `rdf:type` triple.

When the processing of the current element is finished, all child elements that are not text nodes are gathered and processed the same way. The parsing routine will be invoked again, once for each child element, with the *current subject* from the parent's parsing and the depth in the XML tree the parser currently operates on. Currently, there is no depth limitation for eRDF parsing, but there might be one in future to improve parsing performance. For more information on this, see 3.7.

3.3 Triple Store

Integrated into the data management is also the triple store that holds the datasets after they have been parsed from the DOM. When initializing the eRDF parser, a triple store's method to register triples must be passed as the only parameter to the ERDF.init-method. In figure 3.11, you see the initialization method of the Oryx data management. It initializes the eRDF parser with the `DataManager._registerTriple`, which is the triple store's method accepting triples. In 3.2 you have learned how the triples are extracted from the DOM, and all triples known to the triple store can be queried, as you have learned in 3.1.2. To learn more on the internal triple structure, see 3.1.1.1.

Internally, all triples are held in a simple array. Figure 3.12 shows the `_registerTriple`-method as implemented in the data manager. As you see, the triple is simply being added to the top of the `_triples`-array by calling `push` on the array.

Besides `_registerTriple`, the triple store knows the `__synclocal`-method. As you will learn in 3.4, addition and removal of triples is handled by DOM manipulation

```
/**
 * The init method should be called once in the DataManager's lifetime.
 * It causes the DataManager to initialize itself, the eRDF parser, do all
 * necessary registrations and configurations, to run the parser and
 * from then on deliver all resulting triples.
 * No parameters are needed in a call to this method.
 */
init: function() {
    ERDF.init(DataManager._registerTriple);
    DataManager.__synclocal();
}
```

Figure 3.11: Data management and eRDF parser initialization.

```
/**
 * This method is meant for callback from eRDF parsing. It is not to be
 * used in another way than to add triples to the triple store.
 * @param {Object} triple the triple to add to the triple store.
 */
_registerTriple: function(triple) {
    DataManager._triples.push(triple)
},
```

Figure 3.12: Pushing triples to the store.

only. Eventually[13], there is a need to parse the document again when a triple manipulation action transaction succeeded on the DOM. This way, the data management ensures that only what really has a correspondence in the DOM is considered a triple in the store and that there are no inconsistencies with triples that have not yet been rendered back into the XHTML document. So, to eventually parse the created triples back in, call `_synclocal` on your data manager instance.

3.4   Triple Manipulation API

Figure 3.13 gives a brief and not exhausting overview on the methods available in the data management module of Oryx. While the `init`- and `_registerTriple`-method

---

[13] Earlier in the development process, this behavior was fully automated. As it turned out, an automated re-parsing of the document resulted in significant drawback of the data management performance. Moving the responsibility to decide when to re-parse the document into the calling code improved this situation.

have already been described in 3.3, and the `query`-method has been described in 3.1.2, this section will describe the `addTriple`, `removeTriples`, `removeTriple` and `setObject`-methods, how to use them to manipulate eRDF triples, what impact they have on the DOM and what ambiguities one must know about when dealing with them.

All the methods described in this section manipulate the DOM only. As already described in 3.3, it is in the responsibility of the querying and triple manipulating party to synchronize the triple store with the local DOM by calling `_synclocal` on the data manager instance.

```
var DataManager = { [...]

    init: function ( ) { [...] },
    _registerTriple: function ( triple ) { [...] },

    __synclocal: function ( ) { [...] },
    syncGlobal: function ( facade ) { [...] },

    serializeDOM: function ( facade ) { [...] },

    addTriple: function ( triple ) { [...] },
    removeTriples: function ( triples ) { [...] },
    removeTriple: function ( triple ) { [...] },
    setObject: function ( triple ) { [...] },

    query: function( subject, predicate, object ) { [...] }
}
```

Figure 3.13: The triple store interface.

3.4.1   Triple Addition

To simply add a triple to the store, you should use the `addTriple`-method as seen in figure 3.14. Construct a triple and pass it as the only parameter. However, there are certain constraints on the usage of this function:

- The triple subject needs to represent a resource[14].

---

[14] Compare to the first assertion in figure 3.14.

- That resource needs to exist in the current document in order to make statements about it[15].

All literal objects will be grafted into span elements with an appropriate class attribute to produce the same triple again when traversed by the eRDF parser. The literal content becomes the only text node child of the span element. Consider that when there already is a text node in the DOM that has equal content as the literal object, this circumstance is completely ignored. For a detailed discussion of this problem, see 4.1.

3.4.2 Triple Removal

For removal of triples there are two methods:

1. `removeTriple` expects exactly one parameter which is supposed to be a triple as defined in 3.1.1.1. It removes the triple from the DOM and returns true if removal was successful.

2. `removeTriples` expects exactly one parameter, which is supposed to be an array of triples as defined in 3.1.1.1. It tries to remove all triples, gracefully recovers from removal failures and returns an array of triples, that actually have been removed from the DOM.

Triple removal works recursively on the DOM. First, the subject the triple makes a statement about is to be found in the current XHTML document. Therefore, all triples that can be removed from the DOM need to have a resource subject that is embedded into the current document. From there on, all descendant nodes are examined whether they know a class attribute that could produce the specified triple. All elements that match this criterion are removed from the DOM, including all of their children.

Again, there are ambiguities and side effects of triple removal on DOM level. To allow decent removal of DOM contents via triples[16], the deletion of a single triple can result in the removal of the triple's resource object from the DOM. Therefore, triples that were embedded in that part of the DOM will be removed, too. In respect to this side effects, the `removeTriples` method returns an array of the triples that actually have successfully been removed. Consider that triples that should have been removed, but are not in the array of removed triples, could nevertheless have been removed as a side effect of one of the successfully removed triples. See 4.2 for details.

---

[15] Compare to the second assertion in figure 3.14.
[16] A requirement from the Thanis wiki client.

```
addTriple: function(triple) {

    // assert the subject is a resource
    if(!triple.subject.type == ERDF.LITERAL)
        throw 'Cannot add the triple ' + triple.toString() +
            ' because the subject is not a resource.'

    // get the element which represents this triple's subject.
    var elementId = ERDF.__stripHashes(triple.subject.value);
    var element = $(elementId);

    // assert the subject is inside this document.
    if(!element)
        throw 'Cannot add the triple ' + triple.toString() +
            ' because the subject "'+elementId+'" is not in the document.';

    if(triple.object.type == ERDF.LITERAL)

        // object is literal
        DataManager.graft(XMLNS.XHTML, element, [
            'span', {'class': (triple.predicate.prefix + "-" +
                triple.predicate.name)}, triple.object.value
            ]);

    else {

        // object is resource
        DataManager.graft(XMLNS.XHTML, element, [
            'a', {'rel': (triple.predicate.prefix + "-" +
                triple.predicate.name), 'href': triple.object.value}
        ]);
    }

    return true;
}
```

Figure 3.14: The `addTriple`-method.

```
removeTriples: function(triples) {

    // from all the triples select those ...
    var removed = triples.select(

        function(triple) {

        // ... that were actually removed.
        return DataManager.__removeTriple(triple);
    });

    return removed;
}
```

Figure 3.15: The `removeTriples`-method.

```
removeTriple: function(triple) {

    // remember whether the triple was actually removed.
    var result = DataManager.__removeTriple(triple);

return result;
},
```

Figure 3.16: The `removeTriple`-method.

3.4.3 Triple Object Changing

Another useful way to manipulate triples is to change the object of a triple. The `setObject`-method (fig. 3.17) provides a shortcut for this.

```
setObject: function(triple) {

    var triples = DataManager.query(
        triple.subject,
        triple.predicate,
        undefined
    );

    DataManager.removeTriples(triples);

    DataManager.addTriple(triple);

    return true;
}
```

Figure 3.17: The `setObject`-method.

First, the triple store is queried to deliver all triples that share the subject and predicate with the triple you provided. All triples that match this criterion are removed form the DOM. Finally, the one triple you provided is added. You have learned about triple addition and removal in 3.4.1 and 3.4.2.

Ideally, there is only one triple that is affected by this operation. In every case, where there may be several triples describing the same property for the same subject but with a different object, you should not use the `setObject` method, since it affects all of them. Consider that after this method returns, the eRDF parser produces exactly one triple with the subject and predicate you provided, and that that triple has exactly the object that you provided to the method originally. Also make sure you are not changing or accidently removing triples with highly complex content, since, as you read in 3.4.2, triple removal may have side effects.

Another aspect of how this method works is the fact, that the DOM is changed, no matter whether, in the first place, the triple had the object you had liked it to have. And, if eventually the triple you are setting the object for did not change, there may still be side effects that resulted from the initial triple removal.

3.5   Resource Management

Integrated into the data management of Oryx is also a resource management that
handles all server communication. Oryx considers every resource that is stated to
have a `raziel:entry`[17] URL to be available for Oryx resource management.

To get a resource object to work with, simply gather the resource repre-
sentation's id and get a resource object by calling `var r = ResourceMan-
ager.getResource(shape.resourceId)`. The returned resource object knows
methods to save, reload and delete it on the server and in a RESTful manner.
Saving a resource then can be as easy as seen figure 3.18, which is taken from the
Thanis client.

```
ResourceManager.getResource(activity.uri).save();
```

Figure 3.18: Saving a resource in the Oryx resource management.

Reloading and deleting a resource is as easy, you need to call `delete` or `reload` on
the resource object once you instantiated one using the resource id. What happens
exactly is that the resource management sends requests to the `raziel:entry` URL,
and the appropriate data. When saving, the HTTP request is prefixed by the `PUT`
verb, when deleting, it is using the `DELETE` verb, and when reloading, it uses the `GET`
verb in the request.

Creating a new resource is performed by invoking the `__createResource`-method.
An example of a resource creation, which performs a `POST` request on the page-global
`raziel:collection` URL defined in the head of the page, can bee found in figure
3.19

```
var r = ResourceManager.__createResource();
```

Figure 3.19: Creating a resource in the Oryx resource management.

3.5.1   Resource Event Handling

In order to provide a functionality to insert callbacks into the resource manage-
ment[18], the Oryx resource management provides a dedicated event dispatching.

---

[17] Raziel is the namespace of all server relevant Thanis properties.

[18] Which is required for third-party applications, that need to re-parse resource data on changes.

Using the `addListener`-method provided by the resource management, every application can register a callback function on certain resource events. The resource event types need to be composited using bitwise ORs and the resource event types constants visible in 3.20.

```
const RESOURCE_CREATED = 0x01;
const RESOURCE_REMOVED = 0x02;
const RESOURCE_SAVED = 0x04;
const RESOURCE_RELOADED = 0x08;
const RESOURCE_SYNCHRONIZED =  0x10;
```

Figure 3.20: Event types used in resource management

```
ResourceManager.addListener(AWiki.resourceChanged.bind(this),
RESOURCE_CREATED | RESOURCE_SAVED | RESOURCE_RELOADED | RESOURCE_REMOVED);
```

Figure 3.21: Adding a listener on the resource manager of Oryx.

Figure 3.21 shows you how to add a callback on the events of creating, saving, reloading and deleting a resource. This code snippet is taken from the Thanis wiki's source code.

3.6 Designing a Domain Specific Language

During the development process we realized, that the way that querying triples is implemented in the triple store is quite unhandy and difficult to understand when reading source code only. Although the approach using a single `query`-method is sufficient and works well for all use cases, we considered designing a domain specific language for triple querying.

Although there are concrete ideas and even specifications that can be found in the source code, such a language was never fully implemented, due to lack of time and because the general querying technique worked out well in both the Oryx editor and the Thanis Wiki applicaion.

For the sake of completeness, I would like to reference to the source code for more detailed information on how such a language could be implemented and how it should then be used for triple querying and manipulation.

3.7   Performance

Performance of the data management was subject to a lot of improvements over time. In this section, I would like to draw attention on a few concrete techniques that were applied to speed up the Oryx data management:

1. Omitting DOM parts when parsing for eRDF. When the eRDF parser is instructed to parse or re-parse the document, all elements that are not in the XHTML namespace are simply ignored. Since eRDF is only defined on XHTML content, this does not keep triples from being recognized, and since the editor adds lots of SVG content into the document, the parsing effort is minimized dramatically. In addition to that, plugins are introducing lots of DOM extensions[19] [10], which can be omitted the same way, when they are defined in another namespace.

2. Merging distinct rules into one. Several pairs rules that are described separately in [4] have been merged into one, since they are strongly related to each other. Not having to check for ten rules separately speeds up the parsing process.

3. Removing re-parsing automation. The data management in its final implementation does not re-parse the DOM automatically. This is now in the responsibility of any triple querying or manipulating party. Since this avoids unnecessary re-parsing of the DOM, it speeds up the data management.

Other approaches to performance improvement, that were not implemented yet, include:

1. Limiting eRDF parsing depth. The parser could be limited to a certain depth for DOM parsing. This way, wide parts of the DOM could be omitted, which speeds up the parsing process. However, the then unparsed document sections could have produced triples that, with this performance improvement, would remain unknown to the triple store.

2. Allowing entry-points to parse only parts of the local document. The eRDF parser currently parses the whole document only. It would improve the re-parsing performance dramatically if the eRDF parser would be redesigned to allow parsing of DOM subtrees instead of the whole document only. That way, only the parts that really were changed would be parsed.

---

[19] Current plugins are based on the Yahoo! Ext library, which introduces lots of content in the ext (http://b3mn.org/2007/ext) namespace [10].

4. Problems

The main problems with Oryx data management are the ambiguities when adding
triples to or removing triples from the DOM. eRDF seems to be fitting well in setups
where static information needs to be marked up semantically, but where there is the
need to continuously manipulate the eRDF data, there are several problems that
need to be considered. In this chapter I will describe two of those problems, one will
be the addition of triples with literal objects, that could already have corresponding
text nodes in the DOM, the other one is the removal of triples that could have side
effects and cause removal of additional, nested triples.

4.1   Adding Triples to the DOM

```
<div id='someresource'>
    First Activity
</div>
```

Figure 4.1: Triple addition problem, initial state.

Consider the initial situation in figure 4.1. Now, imagine the data manager being
used to add the triple seen in figure 4.2.

```
<#someresource> oryx:description 'First Activity'
```

Figure 4.2: Triple to be added.

There are two possible outcomes, depending on whether the data management respects text nodes that already exist. The first possible outcome, figure 4.3, results from the triple addition when existing text nodes would be respected. The other possible outcome is 4.4, which results in a new, identical text node being generated.

```
<div id='someresource'>
    <span class='oryx-description'>First Activity</span>
</div>
```

Figure 4.3: Triple addition problem, possible outcome 1.

```
<div id='someresource'>
    First Activity
    <span class='oryx-description'>First Activity</span>
</div>
```

Figure 4.4: Triple addition problem, possible outcome 2.

The data management of Oryx chooses to follow the second approach and to ignore existing nodes when adding new triples. This is due to the reduction of complexity and performance consideration. So, using the Oryx data management, you should expect a result as in figure 4.4.

4.2   Removing Triples from the DOM

```
<div id='someresource'>
    <span class='oryx-description'>
        First Activity
        <a href='./execute.php' rel='oryx-executeurl' />
    </span>
</div>
```

Figure 4.5: Triple removal problem, initial state.

When removing a triple from the DOM, there is a problem with possible side effects that may or may not result in the deletion of additional triples. Consider an initial situation as in figure 4.5. Revisit the triple production rules in 3.2. This short example produces two triples, one from the span, the other one from the a element. Now, imagine the data manager being used to remove the triple seen in figure 4.6.

```
<#someresource> oryx:description 'First Activity'
```

Figure 4.6: Triple to be removed.

There are two possible outcomes, depending on whether the data management restructures the DOM to preserve nested statements. The first possible outcome, figure 4.7, results from the triple removal when nested triples are not respected. This example produces no more triples, so two triples have been removed from the DOM. The other possible outcome is 4.8, which results in a restructured DOM, preserving one triple.

```
<div id='someresource'>

</div>
```

Figure 4.7: Triple removal problem, possible outcome 1.

```
<div id='someresource'>

    <a href='./execute.php' rel='oryx-executeurl' />
</div>
```

Figure 4.8: Triple removal problem, possible outcome 2.

In this case, the data management of Oryx chooses to follow the first approach and to ignore nested triples when removing triples. This, again, is due to the reduction of complexity and performance consideration, but also to allow other applications to remove great deals of resource data at once. So, using the Oryx data management, you should expect a result as in figure 4.7.

5. Conclusion

The editor in its final configuration embeds business process data into the web by means of eRDF, a subset of RDF. The deployment of eRDF for data management on the DOM has proven itself suitable for storage of business process data. All properties that were defined in both the BPMN stencil set [9] and for the Thanis server could be expressed well in RDF triples. The stencil set specification [8] even adopted a namespace-driven property definition that directly translates into eRDF schemas.

Although eRDF allows the definition of RDF classes (`rdf:type` triples) to be used with complex data types, this approach was not regarded suitable for the Oryx editor's internal data. Where there were problems with complex data types, such as the bounds of a shape [10], the editor found a solution by serializing the data into string literals, which then could be stored in eRDF triples [10]. This was followed by the need to parse the literals back into the complex types they are containing, which was mastered with little development overhead.

However, there remain some problematic issues with triple manipulation as they are described in 4. These problems and ambiguities remain inherently with not only our implementation, but the eRDF standard itself. Considering the fact that even RDFa would have had the same problems as it uses the data that is visibly rendered on the page to state triples, the only storage technology that would overcome this problems is RDF/XML. The possibility to utilize RDF/XML directly remains to be evaluated.

The Oryx editor in combination with the Thanis server share an AJAX interface that does not provide functionality to bulk update more than one resource at once.

Additionally, asynchronous requests are avoided due to deadlock problems in the server module. Yet, the client-side performance is sufficient in DOM manipulation, serialization, server communication and DOM parsing. Regarding the client-server communication, one must say that a simple AJAX synchronization completely suits our application, as long as the request can be set up asynchronously and processed in parallel by the server.

The only drawback in client-side performance is the need to repeatedly parse the DOM for changes regarding the eRDF triples it generates. When improvement of the clientside performance is desired, the eRDF parser should be changed to allow different entry points to parse only subsets of the document.

6. Outlook

Oryx, in its current state, is suitable for a lot of applications. The editor is capable of designing processes in BPMN, and it is capable of designing processes for concrete back ends with their own stencil set, as proven by support for the Thanis server. The editor is extensible by means of plugins, that could provide additional and specific functionality for each deployment scenario. We have designed a very generic tool that, especially by being a web application, should integrate well in most business process management environments.

In addition to that, Oryx provides the possibility to use the underlying data by means of RDF processing and reasoning on the data. It should be possible to extract RDF/XML data from any Oryx-enabled site by means of GRDDL to then use this data to either build a decent process knowledge base, or to infer new knowledge in the semantic web. RDF data could also be transformed directly into data formats such as XPDL, that are commonly used by workflow engines. A possible deployment of XPDL serialization could be set up using a GRDDL XSL stylesheet to first gather RDF data from Oryx processes, and then use a custom stylesheet to transform this data into XPDL.

Since Oryx uses Ajax to synchronize single resources independently, small changes on the server side can be propagated quickly to the client side. This allows rapid reaction on changes another user could have made to the server and therefore delivers the infrastructure needed to implement collaboration functionality. This allows Oryx to become a collaborative web application in near future.

Another aspect is the usage of Oryx to only visualize data. Since all editing functionality is located in plugins that are fully optional, a read-only view on semantic

data embedded into a website can be achieved very easily. That way, not only processes that are embedded into websites could be visualized, but also ATOM or RSS feeds, RDF graphs or even calendar data and Gantt charts for project management, provided there is a stencil set implemented for them.

Bibliography

[1] Business Process Modeling Notation (BPMN) Specification, Final Adopted Specification. Technical report, Object Management Group (OMG), February 2006. http://www.bpmn.org/.

[2] Mark Birbeck Ben Adida. Rdfa primer 1.0, embedding rdf in xhtml. W3c working draft, W3C, March 2007.

[3] Dave Beckett and Brian McBride. RDF/XML Syntax Specification (Revised). W3c recommendation, W3C, February 2004. http://www.w3.org/TR/rdf-syntax-grammar/.

[4] Ian Davis. Summary Of Triple Production Rules. Technical report, Talis Information Limited, October 2005. http://research.talis.com/2005/erdf/wiki/Main/SummaryOfTripleProductionRules.

[5] Ian Davis. Rdf In Html. Technical report, Talis Information Limited, October 2006. http://research.talis.com/2005/erdf/wiki/Main/RdfInHtml.

[6] Frank Manola, Eric Miller and Brian McBride. RDF Primer. W3c recommendation, W3C, February 2004. http://www.w3.org/TR/rdf-primer/.

[7] Paul V. Biron and Ashok Malhotra. XML Schema Part 2: Datatypes Second Edition. W3c recommendation, W3C, October 2004. http://www.w3.org/TR/xmlschema-2/.

[8] Nicolas Peters. Oryx - Stencil Set Specification. Final bachelor's paper, Hasso Plattner Institute at the University of Potsdam, July 2007.

[9] Daniel Polak. Oryx - BPMN Stencil Set Implementation. Final bachelor's paper, Hasso Plattner Institute at the University of Potsdam, July 2007.

[10] Willi Tscheschner. Oryx - Documentation. Final bachelor's paper, Hasso Plattner Institute at the University of Potsdam, July 2007.