A Design Methodology for MPEG-21 based Digital Policy Management Systems

Frederik De Keukelaere[1], Xin Wang[2], Chris Barlas[3], Thomas DeMartini[2], Saar De Zutter[1], and Rik Van de Walle[1]

[1] Multimedia Lab, Ghent University-IBBT, Sint-Pietersnieuwstraat 41, B-9000 Ghent, Belgium
{frederik.dekeukelaere, saar.dezutter, rik.vandewalle}@ugent.be
[2] ContentGuard, Inc., El Segundo, CA 90245 USA
{xin.wang, thomas.demartini }@contentguard.com
[3] Rightscom, Ltd., London SE1 7HS, UK
chris.barlas@rightscom.com

Abstract. Nowadays, many types of digital content exist and even more ways in which the content can be consumed. Together with these new ways to express digital content came new business models for trading digital content. Digital Rights Management systems were built to govern transactions. This paper discusses a design methodology for creating a Digital Policy Management system, being a part of Digital Rights Management systems. The introduced Digital Policy Management system uses a standards-based approach combining parts of the MPEG-21 Multimedia Framework. For implementing a Digital Policy Management system an architecture of a Governed Execution Environment is defined. It contains a Rights Analysis Tool, which derives the required rights for sets of execution steps using a three-step process of tracking, filtering, and analysis. Finally, this paper shortly discusses two application scenarios: Digital Item Processing and AJAX, in which the introduced design methodology can be applied.

1 Introduction

Currently, there are many types of digital content and probably just as many possible ways of describing them and the context in which they can be used. A big challenge caused by this diversity of technologies is the requirement to interoperate between the different ways in which digital content, or its context, is represented, described, identified, and protected. One possible way to tackle the problem is by providing a very precise definition of what exactly constitutes a ‘Digital Item’ (DI). In MPEG-21, DIs are defined as structured digital objects, with a standard representation, identification, and metadata within the MPEG-21 framework [1].

To realize many business cases, multimedia content is combined with a Digital Rights Management (DRM) [2] system. For example, a content distributor might want to give a license to a consumer to play a resource if the consumer paid a required fee. To realize this scenario, it is necessary to have a DRM system capable of performing many functionalities ranging from content safekeeping, license offering, content distribution, secure content consumption, payment to authorization, authentication, encryption/decryption, and many more [3]. This paper will be focusing on a standards-based approach to develop a methodology for designing a part of an MPEG-21-based DRM system: an MPEG-21 based Digital Policy Management (DPM) system. An extensive discussion on the role of standardization in DRM can be found in [4]. DPM focuses on the design, analysis, implementation, deployment, and use of efficient and secure technology that handles digital information in accordance with the relevant rules and policies.

In this paper, policies for the DPM system are expressed by means of permitted interactions declared in licenses. Typical interactions, for which the rights are granted in a license, are playing, copying, modifying, and so on. To express licenses in a standardized form, MPEG developed the Rights Expression Language (REL) [5] and to express the permitted interactions, called RDD ActTypes, MPEG developed the Rights Data Dictionary (RDD) [5].

The main focus of this paper is the creation of a Governed Execution Environment (GEE) [6] for running program code that needs to behave according to a license. This GEE becomes active, inside a secure environment, after the decryption of a content resource has been done. At that point, it is the responsibility of the GEE to check if a program consuming the resource is not violating any of the permissions it has been granted on that resource.

2 Rights Expressions vs. Application Code

This paper provides a design methodology allowing the construction of an MPEG-21 based DPM system that is ‘well-behaving’. In the context of this paper, well-behaving is defined as acting according to the rights granted in the issued licenses. For granting rights, this paper uses fourteen RDD ActTypes which are listed in Table 1. For each definition, the words in italic are further defined in the RDD standard. This results in an unambiguous definition of the RDD ActTypes.

To be able to know if a program is acting according to the issued licenses, it is necessary to deduce the required rights for performing the actions of the program. This is not a straightforward problem because the licenses grant the rights at a different, higher, level than the operations that happen in the program. For example, suppose a right has been granted to *Print* a resource. The act of printing is formally defined in RDD as ‘To *Derive* a *Fixed* and directly *Perceivable* representation of a Resource’. This high-level definition is perfectly interpretable by humans, and there will be little doubt for a human once he has the printed paper in his hands that he actually performed the *Print* act. For computers on the other hand, printing is most often a combination of calling a set of APIs in a certain order to instruct a printer to *Print* letters on a page eventually resulting in a printed paper. Therefore grants are expressed at a different level than the operations in the program and a mapping between both needs to be made in order to be able to create well-behaving applications and hence an MPEG-21 DPM system.

Table 1. RDD ActTypes Supporting REL

RDD ActType	Definition.
Adapt	To *ChangeTransiently* an existing *Resource* to *Derive* a new *Resource*.
Delete	To *Destroy* a *DigitalResource*.
Diminish	To *Derive* a new *Resource* which is smaller than its *Source*.
Embed	To put a *Resource* into another *Resource*.
Enhance	To *Derive* a new *Resource* which is larger than its *Source*.
Enlarge	To *Modify* a *Resource* by adding to it.
Execute	To execute a *DigitalResource*.
Install	To follow the instructions provided by an *InstallingResource*.
Modify	To *Change* a *Resource*, preserving the alterations made.
Move	To relocate a *Resource* from one *Place* to another.
Play	To *Derive* a *Transient* and directly *Perceivable* representation of a *Resource*.
Print	To *Derive* a *Fixed* and directly *Perceivable* representation of a *Resource*.
Reduce	To *Modify* a *Resource* by taking away from it.
Uninstall	To follow the instructions provided by a *UninstallingResource*.

3 Behavior of Static and Dynamic Applications in DPM Systems

3.1 Static Applications

For an application in which all code is known before run-time, called a *static application* in this paper, the implementer understands, in advance, how the code interacts with DIs. Being well-behaved is simply a matter of finding the appropriate points in the implementation during its interaction with a DI to check for the rights that map to that interaction.

The difficulty to find the appropriate points for incorporating rights checks can be illustrated with the following example. Consider an application that handles a DI containing a video resource. The application adapts the resource by reducing its quality, storing the reduced version in memory, and finally writing the adapted version back to disk. At first, possible points for incorporating the rights checks seem to be located before the adaptation of the resource and before the storing to the disk. However, there is only one appropriate point to incorporate the rights check and that is before storing to the disk. Such a check could not be done earlier (for example, before the adaptation), since the application could have the combined right to store an adapted version but not the combined right to play an adapted version. Hence in this case the appropriate point in time to check for a ‘store adapted resource’ right is just before the storing of the adapted resource. A good methodology when trying to identify the points for incorporating the rights checks into application code is to wait until the results of the program become ‘visible’ [7] to the outside world, i.e., the world outside of the DRM and hence the DPM system. In the example, this means waiting until the result of the adaptation is stored to disk. Other common examples are before playing resources, before printing resources, and so on.

3.2 Dynamic Applications

A *dynamic application*, i.e., an application in which *not* all code is known before run-time, does not allow the preprocessing of the application code to incorporate rights checks. This is because the implementer of the application does not know what code will be executed at the time of writing the application. A widely-used example of such an application is a web browser with scripting support. The implementer of the web browser writes the code for rendering the HTML data and the execution environment for executing the downloaded scripts, i.e., the dynamic code. Since the implementer of the execution environment for the scripts does not know what actions the scripts will perform, he cannot incorporate the required rights checks.

One possible way to tackle this problem would be to look at the script (or any program) that is dynamically loaded as being a static application. An analysis of that static application can be realized using the methodology described above and rights checks could be manually incorporated beforehand in the dynamically loaded code to make the application well-behaving. This methodology will work if the author of the dynamic code also has the intention to make his code well-behaved. However, in an internet scenario where the consumer of the dynamic code, for example, of the web page, does not know if the author of the dynamic code has incorporated the appropriate rights checks, this approach will fail. In other words, when counting on the goodwill of the author of the dynamic code, it is not possible to assure that the dynamic application will be well-behaving in all circumstances.

An alternative to the previous solution is extending the execution environment for the dynamic code to become a GEE in which rights checking is done at run-time by analyzing the different steps in the code and mapping groups of instructions to RDD ActTypes. By incorporating rights checks at run-time it is possible to assure that the dynamic application is well-behaving at all times. How such a GEE can be designed, is discussed in the following sections.

4 An architecture for a governed execution environment

In Fig. 1, we define an architecture for creating a GEE. This environment expects two inputs. The first input is the dynamic code that needs to be executed. This code can be any type of code as long as the APIs, used by the code, are known in advance. The APIs need to be known in advance because an analysis of those will be done at design-time of the GEE. The analysis of APIs will be discussed in Section 6. In the examples throughout this paper, ECMAScript programs accessing and manipulating XML nodes using the Document Object Model (DOM) [8] APIs are used as dynamic code. The second input to the GEE is a license. This paper uses REL licenses which grant RDD ActTypes as the permitted interactions with resources. Note that the methodology discussed in the following sections is applicable to both static and dynamic applications using a broad range of APIs. Dynamic applications using XML-

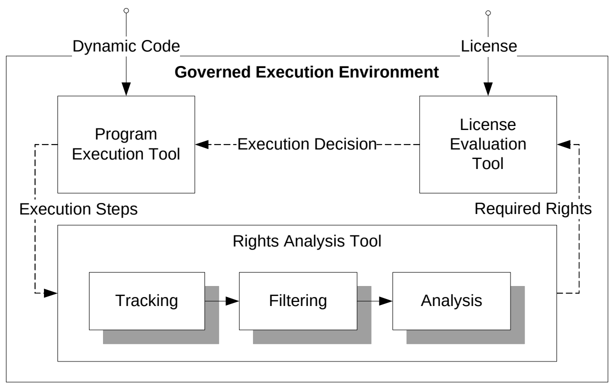

Fig. 1. Architecture for a Governed Execution Environment

manipulating APIs are used here as an example to illustrate the usability of the proposed methodology in internet environments.

The introduced architecture is composed of several different tools for supporting rights checking in dynamic applications: the Program Execution Tool (PET), the Rights Analysis Tool (RAT), and the License Evaluation Tool (LET).

The PET is responsible for executing the code. During the execution of the code, the PET provides execution steps to the RAT. How this information is generated and handled is discussed in Section 5.1.

The RAT is responsible for deducing the required rights for executing a program. It takes the different execution steps from the PET as input and generates the required RDD ActTypes as output. From input to output, a three-step process of tracking, filtering and analyzing execution steps is used. The RAT is discussed in Section 5.

The LET checks for the right to perform the actions defined in the required RDD ActTypes. To realize this, it creates an REL Authorization Request [5] combining the output of the RAT and additional context information such as the current time and date, usage history, and so on. Afterwards, the Authorization Request is evaluated against the licenses provided to the GEE. The LET gives an answer to the following question: ‘is the PET (on behalf of the user, the device or the underlying code) authorized to perform the Execution Steps given the deduced set of required RDD ActTypes and the rights granted in the REL licenses?’

In Fig. 1 the different tools are connected by a dashed line, this illustrates that the different tools in the GEE do not necessarily have to reside on the same terminal. For example, they could be implemented in a distributed way using web services. The main requirement for creating a GEE is that the connection between the different tools is done in a secure way such that the data that is exchanged between the tools cannot be tampered with.

In order to build a more complete DRM system with the GEE, it is possible to use MPEG-21 Intellectual Property Management and Protection (IPMP) [9]. In such a system, the tools introduced above (the RAT, LET, and PET), could be registered through an IPMP Tool Registration Authority [10]. This achieves interoperability in the acquisition of the tools and the messages transmitted between the tools. How this would be done in practice, is out of the scope of this paper, and will not be discussed.

5 Rights Analysis Tool

The RAT is a vital part for creating a GEE and hence for creating well-behaving applications. For the RAT, we distinguish three steps: tracking, filtering, and analysis. This section gives an overview of the functionality of the RAT using an ECMAScript example as listed in Listing 1. More complex situations exist but are out of the scope of this architectural paper.

Listing 1. ECMAScript code removing Chapter One from the book

```
var registry = DOMImplementationRegistry.getDOMImplementation("LS");
var builder = registry.createLSParser(
                         DOMImplementationLS.MODE_SYNCHRONOUS,null);
var did1 = builder.parseURI("DID1.xml");
var book=did1.getFirstChild().getFirstChild();
var chapter1=book.getFirstChild();
book.removeChild(chapter1);
var domWriter = registry.createLSSerializer();
domWriter.writeToURI(did1,"DID1.xml");
```

Listing 2. DID1.xml: A DI representing a book

```
<?xml version="1.0" encoding="UTF-8"?>
<DIDL xmlns="urn:mpeg:mpeg21:2002:02-DIDL-NS">
  <Item id="book">
    <Item id="chapter1">
      <Component>
        <Resource mimeType="text/plain">
           Chapter One - revision 1.
        </Resource>
      </Component>
    </Item>
  </Item>
</DIDL>
```

The example in Listing 1 uses the DOM APIs to load a DI called `DID1.xml` (see Listing 2) and manipulates that document by removing the `Item` node containing Chapter One. Finally, it overwrites the original XML document with the newly created XML document. The output of the RAT will be the set of required rights for the different XML nodes of the document. This example will look at the required rights for the `Item` with the `ID` `book` and the `Item` with `ID` `chapter1`.

5.1 Tracking

In the first step of the RAT, the execution steps of the code are converted to tracked information usable for rights analysis [11]. The tracking process breaks down the original code into elementary actions, for example, `did1.getFirstChild().getFirstChild();` is broken into two calls of `getFirstChild` on each of its objects. To be able to track the nodes throughout the execution steps, each node is initially given a unique `ID`. From that point on, any manipulation of the node results in a new `ID`. The resulting tracked information can be found in Table 2. Note that the first column is generated in the next step and can be ignored for the time being.

Table 2. (Filtered) Tracking Information of Listing 1

F1	ID	Method and Parameters	Original ID	New ID
	#document_3	getFirstChild		DIDL_6
X	DIDL_6	getFirstChild		Item_9
X	Item_9	getFirstChild		Item_11
X	Item_9	removeChild(Item_11)	Item_9	Item_17

5.2 Filtering

In the second step of the RAT, the tracked information is filtered. This filtering process retains tracked data relevant for the analysis of the required rights for a certain XML node. Suppose it is needed to know what rights are required for the node represented by the variable `book`. At the end of the tracking step, the `book` variable points to the `Item` with the XML `ID` `book` and tracked `ID` `Item_17`. To filter out the relevant information for `Item_17`, it is possible to use a backtracking algorithm, called filtering algorithm F1, to the point where `Item_17` first appeared. Backtracking of `Item_17` shows that `Item_17` was created from `Item_9`, which was first accessed from `DIDL_6`. As a result from the filtering process, the first line of the tracked information is no longer considered because it is not related to the `Item` identified by the `book` variable. The tracked data that will be considered for analysis is indicated with an `X` in the first column, `F1` of Table 2. Note that this is an example where filtering is rather straightforward. More complex filtering situations exist but are out of the scope of this architectural paper.

5.3 Analysis

The final step in the RAT is the analysis of the filtered data. For this process it is important to know that rights should be evaluated with a certain node in mind. In addition, it is necessary to map APIs to the RDD ActTypes (see Section 6).

The situation in which one line of tracked information can result in different required RDD ActTypes for the different nodes involved can be illustrated using the last entry, the `removeChild`, of Table 2. For the XML node `book`, with `ID` `Item_9`, the RDD ActType definitions *Reduce* and *Diminish* are potential candidates for required rights for this step. The same line of code requires the *Delete* RDD ActType for the XML node `chapter1` (with `ID` `Item_11`). Therefore, the output of the RAT will always contain a combination of `ID` and required rights.

To further simplify the required rights for the `book`, it is necessary to look at more context information. Based on the definitions of the RDD ActTypes the choice between *Reduce* and *Diminish* is dependent on the availability of the original resource at the end of the execution. In the example, there is only one resource left at the end of the execution (since `DID1.xml` is overwritten see Listing 1 `writeToURI`). Thus, the *Diminish* right is not applicable. Finally, the RAT will report that the *Delete* RDD ActType is required for `chapter1` and the *Reduce* RDD ActType is required for `book`.

6 Analysis of APIs based on a decision tree for RDD ActTypes

Until now in the discussion about the decision making process for deriving required RDD ActTypes, we have been silent on how to find out what are the potential rights that might be triggered by a certain API call. In this section, we introduce a methodology to analyze an API based on a decision tree for RDD ActTypes (see Fig. 2). Note that the tree in Fig. 2 does not contain all RDD ActTypes listed in Table 1. The RDD ActTypes that are not listed can be considered as stand-alone RDD ActTypes and can be evaluated separate from the ones in the tree.

To be able to build a GEE, it is necessary to know the APIs that will be used by applications running in the GEE. This requirement is met both in the static and dynamic applications (as introduced in Section 2). For static applications, the APIs are known before the source code is compiled to an executable and hence also known before the analysis. In this case, analysis of the API can be done before the rights checks are incorporated in the code. To be able to build a PET for dynamic code, it is necessary to know the APIs that the dynamically downloaded code will use. If this would not be the case, it would be impossible to develop an execution environment that would execute the downloaded code. Therefore, the APIs will also be available before execution time to perform an analysis for deducing the RDD ActTypes.

The analysis of an API needs to be performed on the API calls that potentially cause a change in the status of the nodes that need to be analyzed. Hence, the getters only need to be tracked to generate information about the origin of nodes. They do not potentially trigger any required rights. Hence only the setters will be further discussed. The analysis of a setter API call is done using the decision tree displayed in Fig. 2. It is based on the definition of the setter API call and is performed on the nodes involved in its execution. A DOM API call can be seen as:

```
return value = called node.method(parameters)
```

As discussed earlier, deciding what rights are potentially required for a certain API call depends on the node that is considered while analyzing. Therefore, the API needs to be separately analyzed for each node used in the API call, thus for `return value`, `called node`, and each of the nodes passed as `parameters`.

For example, consider the `removeChild` API call from the DOM API (see Fig. 3). In this case, there are three nodes for which potentially rights can be required. The first node is the `oldChild` parameter. The second node is the `called node` on which the `removeChild` method is called. The third node involved in this API call is the `return value`. This is the same node as the `oldChild` parameter, and therefore requires the same rights.

The analysis of the required rights for the nodes can be done using the decision tree for RDD ActTypes. This decision tree is designed by comparing the state of the nodes before executing the API call with the state after executing the API call. Since there might not be a conclusive answer to all of the questions in the tree at the point of evaluating, several routes can potentially be followed. As an example, the required rights for the node on which the `removeChild` method is called, i.e., the `called node` are deduced in the next paragraphs.

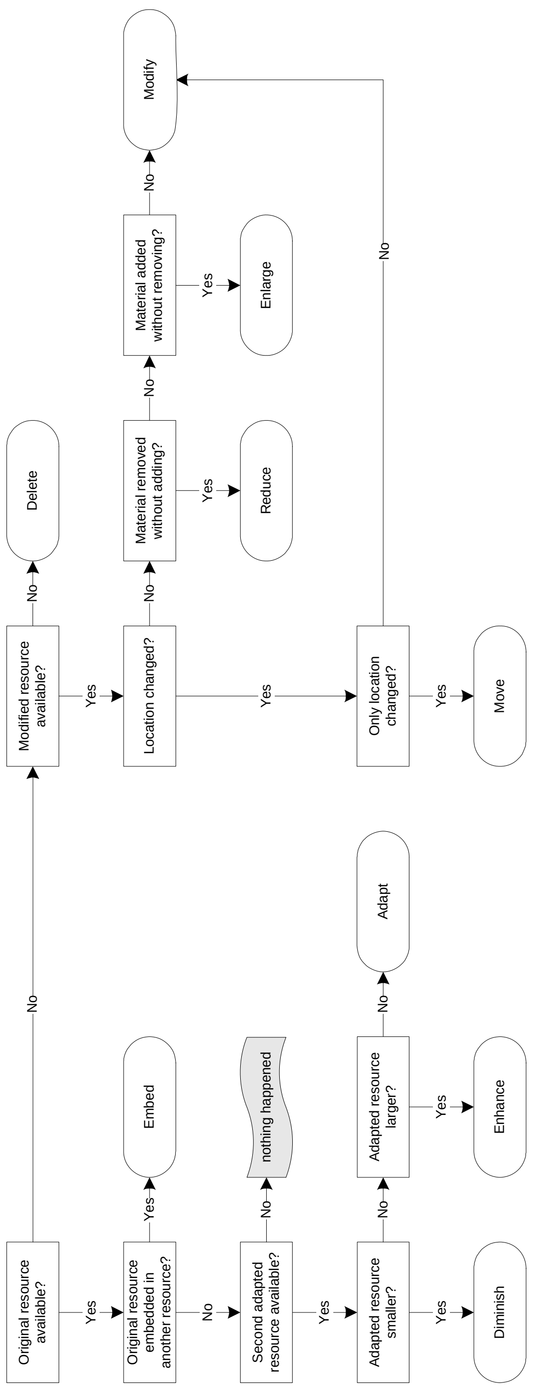
Original resource available?
Yes
No
Original resource embedded in another resource?
Yes
Embed
No
Second adapted resource available?
No
nothing happened
Yes
Adapted resource smaller?
Yes
Diminish
No
Adapted resource larger?
Yes
Enhance
No
Adapt
Modified resource available?
No
Delete
Yes
Location changed?
No
Material removed without adding?
Yes
Reduce
No
Material added without removing?
Yes
Enlarge
No
Modify
Yes
Only location changed?
Yes
Move
No

Fig. 2. Decision Tree for RDD ActTypes

Table 3. RDD ActTypes potentially triggered by removeChild

Node	RDD ActType	Condition
`oldChild`	*Delete*	`called node` overwritten
		`return value` not used
`called node`	*Reduce*	`called node` overwritten
	Diminish	`called node` not overwritten

```
removeChild
  Definition
    Removes the child node indicated by oldChild from the list of
    children, and returns it.
  Parameters
  oldChild of type Node
    The node being removed.
  Return Value
  Node
    The node removed.
```

Fig. 3. Definition of the removeChild Method

The first question in the decision tree is ‘is the original resource available after executing the API call?’ Suppose the original node (i.e., the node before executing the API call) is overwritten during the execution of the program, the answer to this question is ‘no’. The next question is ‘is there a modified resource available?’. Since there is material removed from the original node, there is a modified node available; hence the answer is ‘yes’. The next question is ‘did the location of the resource change?’. The answer to this question is ‘no’, because there was only a child removed from the `called node`, nothing happened to the location of the `called node`. The next question is ‘is there material removed without material being added?’. The answer to this final question is ‘yes’. Therefore, the *Reduce* RDD ActType could be required by executing this API call.

Suppose the node was not overwritten during the execution of the program. In that case the answer to the first question would be ‘yes’. The following question is ‘is the original resource embedded in another resource?’. The answer to this question is ‘no’ because none of the actions described in the *Embed* definition were performed. There was only material removed from the `called node`. The next question is ‘is there a second adapted resource available?’. Since the newly stored node is a newly created node which is derived from the original `called node`, there is a second adapted node of the original `called node`. Therefore the answer to this question is ‘yes’. The answer to the next question ‘is the adapted resource smaller than the original?’ is ‘yes’, because material has been removed from the original to derive the second adapted node. Therefore, the *Diminish* RDD ActType could be required by executing this API call.

In addition to the analysis from the `called node` point of view, it is also necessary to perform the analysis from the `oldChild` point of view. This can be realized in the same way as discussed above. An overview of the RDD ActTypes and the associated conditions that can potentially be triggered by `removeChild` is given in Table 3. Other API calls can be analyzed using the same methodology.

In this section, it was discussed how it is possible to map API calls to the basic set of RDD ActTypes of Table 1. However, RDD provides the possibility to extend this set of basic ActTypes to create highly specialized combinations of ActTypes [12]. How an API analysis can be realized for those specialized RDD ActTypes has not been investigated yet. Therefore, in this paper, we will focus on the basic RDD ActTypes.

7 Possible application scenarios

The first application scenario is based on the MPEG-21 Digital Item Processing (DIP) [14] technology. The main goal of DIP is to extend the concept of a DI to include programmability into DIs. The relationship between DIs and DIP is similar to the relationship between XHTML and JavaScript. DIP allows the declaration of dynamic behavior in DIs. DIP is largely based on ECMAScript and DOM for manipulating and handling DIs. Since DIP can be considered a dynamic application, it is possible to build a GEE for DIP based on the design methodology described above. Such a governed environment for executing DIP applications can be used to build a larger MPEG-21 based DRM system.

The second application scenario comes from a new trend in internet applications to make websites more dynamic with the ultimate goal to result in a local-client experience for the end-user while interacting with the web. One of the key technologies in realizing this concept is Asynchronous JavaScript + XML, better know as AJAX. Major players in internet applications are currently using AJAX technologies to increase the performance and interactivity of their web sites. For example, Google Mail, MSN Virtual Earth, and Yahoo! Instant Search are making extensive use of AJAX. Since AJAX is based on DOM and JavaScript and considering the interest in AJAX, it might prove to be an interesting application for our introduced technologies.

8 Conclusions

In this paper, we introduced a design methodology for building MPEG-21 based DPM systems. Such DPM systems allow the creators of software that handles content with associated licenses, to write applications that treat content according to the rights granted in the licenses.

For creating DPM systems, we distinguished two classes of applications: static applications and dynamic applications. In the former, all code is known in advance, in the latter only the APIs that will be called are known in advance. To create a DPM system for dynamic applications, we described a design methodology for a GEE incorporating a RAT. A detailed discussion of the tracking, filtering, and analyzing algorithms used in the RAT, was given. To create those algorithms, we developed a decision tree for determining the required RDD ActTypes.

Although the design methodology can be applied to various APIs, we applied it to the DOM API. Since the DOM API is used on a large scale in applications using XML, the results of this chapter can be applied without modification in many applications, for example, in Digital Item Processing and AJAX.

Acknowledgements

The research activities described in this paper were funded by Ghent University, the Interdisciplinary Institute for Broadband Technology (IBBT), the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT), the Fund for Scientific Research-Flanders (FWO-Flanders), the Belgian Federal Science Policy Office (BFSPO), and the European Union.

References

1. De Keukelaere, F., Van de Walle, R.: Digital Item Declaration and Identification. In: Burnett, I., Pereira, F., Van de Walle, R., Koenen, R. (eds.): The MPEG-21 Book. John Wiley & Sons Ltd, Chichester (2006) 69-116
2. Guth, S.: A Sample DRM System. In: Becker, E., Buhse, W., Günnewig, D., Rump, N. (eds.): Digital Rights Management - Technological, Economic, Legal and Political Aspects. Lecture Notes in Computer Science, vol. 2770. Springer-Verlag, Berlin Heidelberg New York, (2003) 150-161
3. Gooch, R.: Requirements for DRM Systems. In: Becker, E., Buhse, W., Günnewig, D., Rump, N. (eds.): Digital Rights Management - Technological, Economic, Legal and Political Aspects. Lecture Notes in Computer Science, vol. 2770. Springer-Verlag, Berlin Heidelberg New York, (2003) 16-25
4. Rump, N.: Can Digital Rights Management Be Standardized? IEEE Signal Processing Magazine, vol. 21, no. 2 (2004) 63-70
5. Wang, X., DeMartini, T., Wragg, B., Paramasivam, M., Barlas, C.: The MPEG-21 Rights Expression Language and Rights Data Dictionary, IEEE Transactions on Multimedia, vol. 7, no. 3 (2005) 408-417
6. De Keukelaere, F., DeMartini, T., Wang, X., De Zutter, S., Lerouge, S., Van de Walle, R.: An Architecture for run-time analysis enabling rights checking in dynamic applications, Workshop on Image Analysis for Multimedia Interactive Services (2006).
7. Rust, G., Bide, M.: The <indecs> metadata framework (2000)
8. World Wide Web Consortium: Document Object Model Level 3 Version 1.0 (2004)
9. Huang, Z.Y., Shen, S.M., Ji, M., Senoh, T.: Management and Protection of Digital Content with the Flexible IPMP Scheme - MPEG-21 IPMP, Visual Communications and Image Processing (2005) published on CD-ROM
10. Lauf, S., Rodriguez, E.:IPMP Components. In: Burnett, I., Pereira, F., Van de Walle, R., Koenen, R. (eds.): The MPEG-21 Book. John Wiley & Sons Ltd, Chichester (2006) 117-138
11. De Keukelaere, F., DeMartini, T., Bekaert, J., Van de Walle, R.: Supporting rights checking in an MPEG-21 Digital Item Processing environment, International Conference on Multimedia & Expo (2005) published on CD-ROM
12. ISO/IEC: ISO/IEC 21000-7:2004/Amd 1 Information technology -- Multimedia framework (MPEG-21) -- Part 7: DIA Conversions and Permissions (2006)
13. World Wide Web Consortium: XML Information Set (2004)
14. De Keukelaere, F., De Zutter, S., Van de Walle, R.: MPEG-21 Digital Item Processing, IEEE Transactions on Multimedia, vol. 7, no. 3 (2005) 427-434