DSNotify – Detecting and Fixing Broken Links in Linked Data Sets

Bernhard Haslhofer
University of Vienna
Department of Distributed and Multimedia Systems
Email: bernhard.haslhofer@univie.ac.at

Niko Popitsch
University of Vienna
Department of Distributed and Multimedia Systems
Email: niko.popitsch@univie.ac.at

***Abstract*—The Linking Open Data (LOD) initiative has motivated numerous institutions to publish their data on the Web and to interlink them with those of other data sources. But since LOD sources are subject to change, links between resources can break and lead to processing errors in applications that consume linked data. The current practice is to ignore this problem and leave it to the applications what to do when broken links are detected. We believe, however, that LOD data sources should provide the highest possible degree of link integrity in order to relieve applications from this issue, similar to databases that provide mechanisms to preserve referential integrity in their data. As a possible solution, we propose DSNotify, an add-on for LOD sources that detects broken links and assists the data source in fixing them, e.g., when resources were moved to other Web locations.**

I. Introduction

As part of the Linking Open Data (LOD) initiative, numerous data providers have started to expose their data as structured data on the Web and to interlink them with data from other data providers. They now form the so called *Linked Data cloud*[1]. Prominent LOD data sources are DBpedia [1], which is the structured version of Wikipedia, the BBC Music Portal[2], or the Swedish Library Union Catalogue [2]. They have in common that besides providing human-friendly (HTML) presentations, they also expose machine-processable resource descriptions (RDF) on the Web that can be accessed and processed by clients by dereferencing their HTTP URIs.

Links between resources in different data sources take a central role in the LOD approach. They are usually created either manually or by using semi-automatic link discovery tools (see e.g., [3]). Figure 1 shows a resource that represents a band and is exposed by the BBC Music LOD source (*link source*). Amongst other descriptive data, it contains a typed *link* (owl:sameAs) to a resource in the DBpedia LOD source (*link target*). An application, which processes the data provided by the BBC Music LOD source, could follow the link and retrieve further information about this band from DBpedia, such as the band's members. Hence, a major benefit of the LOD approach is that machines can use the Web as humans already do: they can easily access Web resources by their URIs and follow available links to retrieve further contextually relevant data.

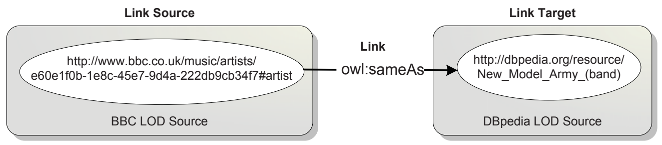
Link Source
http://www.bbc.co.uk/music/artists/
e60e1f0b-1e8c-45e7-9d4a-222db9cb34f7#artist
BBC LOD Source
Link
owl:sameAs
Link Target
http://dbpedia.org/resource/
New_Model_Army_(band)
DBpedia LOD Source

Fig. 1. Sample link between BBC Music and DBpedia data.

One problem with the LOD approach of data management is that applications face the same difficulties as humans do when browsing the Web: links between resources may become broken resulting for example in HTTP error responses or no response at all (see e.g., [4]). Reasons for this can be manifold: web servers may be down or resources might be removed or moved to other locations. While the problem of broken links is annoying for human end-users and the typical reaction is using the browser's back button or manually looking up the intended link target using a web search engine, this is much harder for machines.

In the context of LOD, we define *link integrity* as a qualitative property that is given when all links within and between a set of data sources are valid and deliver the result data intended by the link creator. Traditional data management systems (e.g., RDBMS), have built-in mechanisms (e.g., referential integrity) that preserve such kind of integrity. Unfortunately, we cannot apply these mechanisms for LOD data, because there is no single instance that has full control over the whole data set. Therefore, we require an alternative solution for dealing with the broken link problem. In particular, we need a solution that (i) detects broken links between resources and (ii) provides support for fixing those links.

Our central contribution is a solution that can detect broken links in LOD data sets and assists in repairing them by proposing fixes for broken links. As a proof-of-concept, we have implemented these two contributions in an initial prototype called *DSNotify*. Comparable to iNotify[3] in the Linux kernel, it can be used by an LOD source to get informed about *create*, *remove*, and *move* operations in remote LOD sources.

[1]http://www4.wiwiss.fu-berlin.de/bizer/pub/lod-datasets_2009-03-27.html
[2]BBC Music: http://www.bbc.co.uk/music/

[3]see http://www.linuxjournal.com/article/8478

II. Broken Links in LOD Datasets

The preferable method to deal with the broken link problem is to *avoid* that links break in the first place. Unfortunately, this strategy can only be applied if both, links and resources, are under full control of one single system (like e.g., in an RDBMS), which is not the case in Web environments. Another strategy — the status-quo in the LOD context — is to *ignore* the problem and shift it to higher-level applications that process the exposed data. Our approach is to *fix* broken links wherever possible. In the following, we elaborate on possible types of broken links, discuss strategies for fixing such links and analyze the implications on the DSNotify design.

A. Possible Types of Broken Links

We speak of a *broken link* whenever an HTTP GET request executed against the HTTP URI of the link target does not resolve to a resource description (RDF) but delivers an error response. We can classify the possible types of broken links according to the type of operations that cause links to break:

1) *Removed link targets*: The target of a link is removed permanently from the Web or is not reachable anymore. Reasons for this include that the hosting server is switched off or that the respective resource has been removed from the server. In the latter case, an HTTP Server would return an HTTP error response, such as `404 Not Found`, when an application looks up the link target's URI.
2) *Moved link targets*: The target of a link is not reachable anymore because it has been moved to another Web location, i.e., it is available via another HTTP URI. This happens for instance, whenever an organization changes its domain name (e.g., from dbpedia.org to lodpedia.org) or relocates resources (e.g., from http://dbpedia.org/resource/New_Model_Army_(band) to http://dbpedia.org/resource/band/New_Model_Army).

B. Possible Reactions on Broken Links

In order to preserve link integrity, an LOD data source can react differently on each type of broken link: a possible reaction for the first case (removed link targets) would be to remove broken links. In RDF this means to remove all statements containing the removed link target.

Reacting on moved link targets is more complicated but equally important. Assuming that the resource identifier of the moved link target refers to the same[4] real-world entity as the resource identifier of the original link target, the operation *move* can also be interpreted as *remove* and *add*. The first step is then the same as in the first case: removing the affected links. In a second step, an LOD data source can fix the link by updating the link target with an HTTP URI that identifies the *same* resource as the removed link target.

[4]In the (Semantic) Web community, the notions of identity and equality are still open issues (cf. [5]). In this work, we conceive equality as a subjective decision that depends on the context of an application.

C. Design Considerations

Regarding the Linked Data cloud, we can observe that a *local* LOD data source is always interlinked with $0 \ldots n$ *remote* data sources. For DSNotify we can identify two possible application scenarios: either it is installed as an add-on to a local LOD data source, observes link targets in $0 \ldots n$ remote sources, and *notifies* the local data source whenever a link target becomes unavailable, i.e., is removed or moved. Alternatively, the remote LOD data sources integrate the DSNotify add-on, which observes modifications in the (local) data sets and sends *notifications* to *subscribed* LOD sources (see e.g., [6]). Although the second alternative is more economical considering network and storage costs, we currently focus on the first scenario because we cannot influence the behaviour of remote LOD data sources maintained by other institutions.

The main challenge lies in the decision, whether a link target resource has been removed permanently or moved to another Web location. Since the identity of a resource is determined by its URI identifier, which changes whenever a resource is moved, we cannot rely on that identifier for deciding on the equality of resources but must apply *heuristic methods* based on a resource's RDF description. If the RDF description of a moved resource remains unchanged or contains some kind of structured identifier (e.g., ISBN number), the heuristics could be a simple string comparison and the chance to identify moved resources is very high. If a resource's RDF description comprises mainly low-level metadata (e.g., file-size, mime-type) and maybe a reference to some multimedia object (e.g., an image or a video), more advanced heuristics including different kinds of feature extraction mechanisms are required. In such cases, human intervention may be necessary in order to decide whether two HTTP URI identifiers refer to the same resource, i.e., that a link target resource has been moved. So it must be possible to tailor the heuristics to domain-specific requirements and consider the possibility to include user feedback/intervention in the DSNotify design.

III. Approach

Our proposed solution to the problem of broken links in LOD data sets is called DSNotify. Figure 2 depicts an overview of its architecture: we consider an LOD source (shown on the left hand side) that is linked to other LOD sources (right hand side). This source knows how to update its data and needs to fix broken links occurring when the linked LOD sources are updated.

At the very core of DSNotify lies an indexing infrastructure: A *monitor* (e.g., a web crawler) accesses considered LOD *items*, extracts *features* describing an item and derives an *item representation* (including an item's URI and its feature vector) that is stored in an *index*. The monitor detects what items were *created*, *removed* or *modified* by consulting the index. Detected events are written to a central *event log* and consecutively result in notifications sent to registered applications[5].

[5]Note that this infrastructure would already be sufficient to provide a *broken link detection* service to LOD applications.

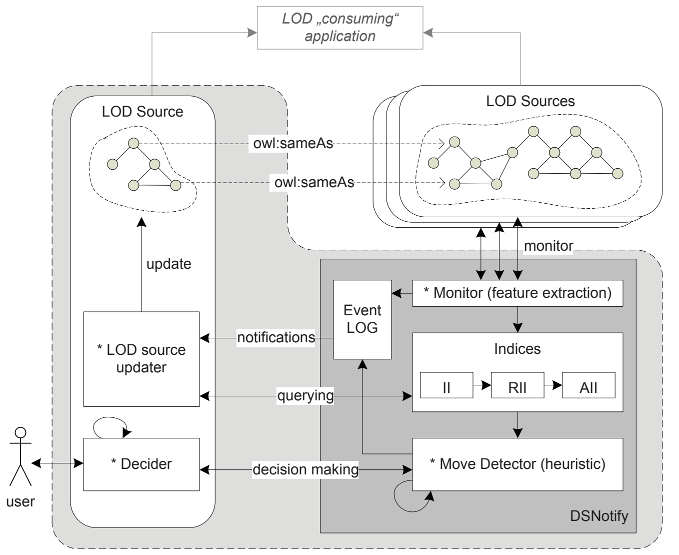

Fig. 2. DSNotify architecture. Components that require some domain-dependent implementation are marked with an asterisk.

A. Feature Vectors Represent Items

The central task of DSNotify is to detect whether items that are not reachable under a particular URI anymore were *moved* and are now reachable under a different URI. In order to do this, DSNotify does not index the content of its considered items but rather extracts a *feature vector* from an item's content and its accessible meta data. This vector consists of *features* that basically constitute name/value pairs (e.g., selected property-values of the corresponding LOD item). The name of a feature is represented by a URI, its value is implemented by some *data type*. DSNotify provides standard feature implementations for XML schema data types (e.g., *string*, *integer* or *boolean* features), but as the data type of features may be largely domain-dependent we also permit to plug-in custom data types like for example a *histogram* type for images. For the reasons explained below, it is required that features are comparable to each other.

Possible features in the case of the example shown in Figure 1 could be the band's name in DBpedia (e.g., `rdfs:label = "New Model Army"`) and/or the current and past members of that band (e.g., `dbpprop:currentMembers = dbPedia:Justin_Sullivan`). In general, these properties are sufficient to uniquely identify a band.

B. Indices and Move Detection

An item's representation, consisting of its identifier (URI), its feature vector, and some administrative meta data, is stored in an *index* that is periodically updated by the *monitor*. In fact, DSNotify keeps three distinct indices: an *item index*, a *removed item index* and an *archived item index*. Item representations are moved between these three indices as illustrated in Figure 3 and explained in the following:

Originally, representations of new items detected by the monitor are stored in the *item index*. Whenever a monitor reads an item, it also extracts the item's feature vector and, if the item is not represented yet, stores the item together with its feature vector in the item index. If the item is already represented in the index but has changed since the last monitoring cycle, the feature vector is updated in the index. For example, at time *t1*, a single item http://dbpedia.org/resource/NewModelArmy(band) is stored in the item index.

When DSNotify detects that an item cannot be accessed any more (because it was removed or moved), the corresponding index entry is not removed but rather moved to the *removed items index* (cf. time *t2*).

Some (configurable) time after this happens, the *move detector* tries to find out what event really took place (*move* or *remove*). It uses a plug-in *heuristic* that compares this item's feature vector to the feature vectors of all *recently added items* by comparing each single feature and combining the results. Our default implementation of this heuristic basically averages the similarity values of all compared features, which might be sufficient for the example mentioned above. For more complex scenarios, advanced strategies may be needed to achieve satisfactory results (e.g., the influence of single features on the total similarity should be configurable).

The result of this comparison is a list of item/probability-value pairs denoting what items *could* be *newer representations* of the considered item. Now it has to be decided whether some item from this list actually *is* a newer representation (thereby triggering a *moved* event) or whether none of them are (thereby triggering a *removed* and a *created* event). Depending on its configuration, DSNotify decides this autonomously (based on threshold values) or sends a *decision request* to some subscribed application that makes the decision (e.g., by incorporating feedback from a human user).

When an item was considered as *moved to a new location*, the corresponding *old* representation is moved from the *removed items index* to a third index called the *archived items index*. Additionally it is linked to its corresponding *newer* representation in the *item index* (cf. time *t3*). Thereby, over time, representations of a particular item form a timely-ordered linked list (see time *t4*: here 3 representations of an item, arranged in a linked list, can be seen).

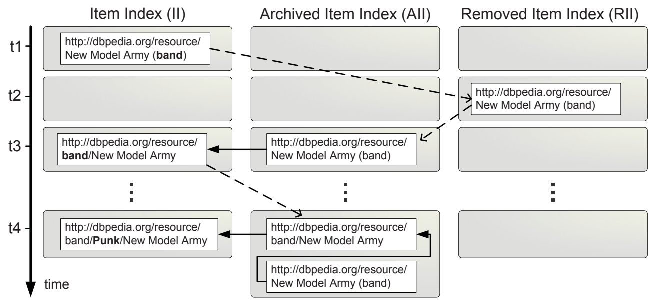
Item Index (II)
Archived Item Index (AII)
Removed Item Index (RII)
t1
t2
t3
t4
time
http://dbpedia.org/resource/ New Model Army (band)
http://dbpedia.org/resource/ New Model Army (band)
http://dbpedia.org/resource/ band/New Model Army
http://dbpedia.org/resource/ New Model Army (band)
http://dbpedia.org/resource/ band/Punk/New Model Army
http://dbpedia.org/resource/ band/New Model Army
http://dbpedia.org/resource/ New Model Army (band)

Fig. 3. Diagram explaining how the three DSNotify indices work together. Item representations are moved between indices (dashed arrows) and archived item representations are linked (solid arrows) to corresponding *newer* representations. Note that some intermediate steps between time t3 and t4 were omitted for simplicity (see text).

C. Feedback and Notification Mechanisms

As depicted in Figure 2, applications may interact with DSNotify in three possible ways: (1) by passively receiving event notifications, (2) by actively querying the DSNotify indices, or (3) by answering decision requests issued by the *move detector*.

In the first case, applications may register for getting event notifications from the central DSNotify event log via various types of communication protocols. Currently we have implemented a simple XML-RPC interface but in principle any kind of Web-based interface is imaginable. DSNotify could, for instance, deliver notifications to subscribing applications using HTTP PUT requests.

The second possibility is that applications actively query DSNotify via its HTTP interface to find out what happened to a particular item identified by a certain URI. DSNotify searches its indices for the passed URI and answers depending on the index it is found in. In case it is found in the *item index* or in the *removed item index*, DSNotify answers with "*existing*" or "*removed*" respectively. In case it is found in the *archived item index*, DSNotify follows the links to the latest item representation and answers with the new URI. Thereby DSNotify may be used as a service that maps historical to recent URIs, for example by applications that do not make use of the notification service and detect that some item is unavailable (e.g., due to an HTTP error response).

In the third case, DSNotify decides between move or remove/create events by consulting a list of item/probability pairs. In some situations this decision can be made automatically with a high probability of correctness: for example, consider that only the URI of an item has changed. In this case, the feature vector comparison of the old and the new item representation should return the highest possible similarity value and – in case there are no *duplicates* with equal feature vectors – it is straightforward to decide that this item was moved. But in other cases (e.g., when the URI and some properties of the item were changed since the last monitoring cycle) it may be difficult or even impossible for DSNotify to make this decision. For this latter case, DSNotify may *outsource* the decision to a subscribed *decision making* application. This application may now decide such "borderline cases" based on its own heuristics, possibly incorporating external knowledge DSNotify is not aware of, or by consulting human users, e.g., in the form of positive or negative feedback. We are well aware that such cases are very likely to occur, therefore this external decision making loop is a core component of DSNotify. By that, an LOD source may decide for itself what strategy to choose to avoid as many *false negatives* (i.e., broken links) and *false positives* (i.e., links that point to wrong data items) as possible.

IV. Implementation

We have implemented a DSNotify Java prototype that includes all the components described in Section III. The index implementations are based on *Apache Lucene*[6], whereas for each index type one can choose between a slow (but persistent) and a fast (in-memory) index. For the archived item index, for instance, the default configuration uses a persistent index. In order to support non-text based content as well, it was necessary to extend Lucene's information retrieval functionality by customizable feature vectors. The LOD data sources to be observed, the feature extraction algorithms to be applied on the data, as well as the heuristics used to detect moved resources can be tailored to domain-specific requirements via external configuration files and plug-in implementations (Java classes).

In the test-runs we have carried out so far, we simulated location and data modifications in available LOD data sets[7] and assured that the behaviour of DSNotify reflects our strategy of detecting and fixing broken links. A comprehensive long-term evaluation on the efficiency and effectiveness of DSNotify in combination with real-world LOD sources will be the next step.

V. Related Work

Early hypertext systems clearly considered link integrity to be an integral feature. But with the application of hypertexts in distributed environments it became more and more difficult to maintain this integrity. Alternative approaches to the Web, which implement more advanced linking concepts and strategies for ensuring link integrity, such as *Hyper-G* [7] or *W3Objects* [8], never made it "out of the lab".

We can divide existing strategies for avoiding and fixing broken links, into the following groups:

- *Indirection* — A layer of indirection is introduced that allows content providers to keep links to their hosted resources up to date. The *PURL*[8] and the *Handle* [9] system, the underlying technologies for *Digital Object identifiers (DOIs)*, are two well known examples for this (see e.g., [10]). Both implement the concept of *Uniform Resource Names (URNs)* to identify resources on the Web and provide services for translating these URNs to resolvable URLs.
  The disadvantage of this strategy is that central services are required for the translation step. Just like it is the case for the domain name service (DNS), clients must be aware of how to address these services and they introduce additional latency when accessing resources. Furthermore, content providers still have to (manually) update the physical address bound to a URN in these services.
- *Redundancy* — Redundant copies of resources are kept and a service forwards referrers to one of these copies as long as at least one copy still exists. However, such services can reasonably be applied only to highly available,

[6]http://lucene.apache.org

[7]For testing purposes we use subsets of the data exposed by OAI2LOD data sources. See: http://www.mediaspaces.info/tools/oai2lod/

[8]The PURL system makes use of the HTTP redirect facility which itself provides only a partial solution for the broken link problem, cf. [8]

unmodifiable data. Examples for such systems include LOCKSS [11] and RepWeb [12].

- *Robust link implementations* — Among the approaches for fixing links automatically, the solution proposed by Phelps and Wilensky [13] is the most relevant in our context: they propose *robust hyperlinks* based on standard Web technologies. The URLs of resources referred to by robust hyperlinks are augmented by a small *signature* composed of terms extracted from the referenced document. When the targeted document is moved, these terms may be used to automatically search for it using a Web search engine and the broken link may be fixed. The authors found out that five terms are enough to uniquely identify a web resource in virtually all cases. A disadvantage of the robust hyperlink solution is that it requires existing URLs to be changed and that it depends on web search engines, therefore being restricted to the indexed part of the Web. Furthermore it is unclear how to extend this method to non-textual resources. Nevertheless we consider a *lexical signature* as a very useful DSNotify feature data type for LOD sources that link to external text documents.
- *Manual / pragmatic approaches* — In this group we summarize strategies of Web users respectively content providers to fix broken links manually. This includes "manually" searching for resources using search engines (basically this is the strategy the above-mentioned robust link idea tries to automate) or direct manipulation of the URL.
  Content providers sometimes make use of automatic link checking software[9] to get informed when broken links occur on their websites. On the other hand, users may utilize services like *ChangeDetection*[10] for monitoring particular Web resources and get informed when they change. Such systems may already be used to solve the broken link detection problem described in Section II but provide no means to support applications in (semi-) automatic link fixing.

VI. Conclusions and Future Work

A major goal of this paper is to create the awareness that when we consider the Web as a novel form of data management as done in the LOD approach, we also have to provide means for preserving link integrity in the exposed data sources, an issue that has already been realized in more established domains (e.g., referential integrity in RDBMS). As a possible solution, we have presented DSNotify, which is an add-on for LOD data sources that can keep links between LOD resources consistent with very little required user input. It allows LOD data sources to react not only on *removed* but also on *moved* link targets in remote LOD sources. The decision whether a link target has been moved is computed heuristically on the basis of domain-specifics features.

DSNotify is highly configurable in terms of its application domain and can therefore serve in many different scenarios. We believe that, when it becomes an integral part of LOD storage solutions, it could help in stabilizing the links between sources and increasing the quality of data provided as part of the Linked Data cloud.

In our future work, we will first perform an evaluation on the efficiency and effectiveness of the DSNotify approach. We will use the currently available DBpedia dumps and set up a testbed comprising several LOD instances, where each instance reflects the state of DBpedia at a certain point in time. We will annotate the exposed data in order to build a benchmark data set and perform a precision-recall analysis. Possible extensions of DSNotify are other types of resource modifications, e.g., modifications within arbitrary RDF data, and, of course, extending this approach to other types of data sources, such as the Web of (HTML) documents or remote (semantic) file systems. We expect to publish a first open source release of DSNotify within the next months.

References

[1] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives, "Dbpedia: A nucleus for a web of open data," in *Proceedings of the 6th International Semantic Web Conference (ISWC)*, ser. Lecture Notes in Computer Science, vol. 4825. Springer, 2008, pp. 722–735.

[2] M. Malmsten, "Making a Library Catalogue Part of the Semantic Web," in *Proceedings of the International Conference on Dublin Core and Metadata Applications*, J. Greenberg and W. Klas, Eds., September 2008.

[3] J. Volz, C. Bizer, M. Geadke, and G. Kobilarov, "Silk - a link discovery framework for the web of data," in *2nd Linked Data on the Web Workshop (LODW2009), co-located with WWW 2009, Madrid, Spain*, 2009.

[4] H. C. Davis, "Hypertext link integrity," *ACM Comput. Surv.*, vol. 31, no. 28, p. 28, December 1999.

[5] H. Halpin, "Identity, Reference, and Meaning on the Web," in *Proceedings of the Workshop on Identity, Meaning and the Web (IMW06)*, 2006.

[6] S. Auer, S. Dietzold, J. Lehmann, S. Hellmann, and D. Aumüller, "Triplify: light-weight linked data publication from relational databases," in *WWW '09: Proceedings of the 18th international conference on World wide web*. New York, NY, USA: ACM, 2009, pp. 621–630.

[7] K. Andrews, F. Kappe, and H. Maurer, "The hyper-g network information system," *Journal of Universal Computer Science*, vol. 1, no. 4, pp. 206–220, 1995.

[8] D. Ingham, S. Caughey, and M. Little, "Fixing the "broken-link" problem: the w3objects approach," *Comput. Netw. ISDN Syst.*, vol. 28, no. 7-11, pp. 1255–1268, 1996.

[9] R. Kahn and R. Wilensky, "A framework for distributed digital object services," *Int. J. Digit. Libr.*, vol. 6, no. 2, pp. 115–123, 2006.

[10] W. Y. Arms, "Uniform resource names: handles, purls, and digital object identifiers," *Commun. ACM*, vol. 44, no. 5, p. 68, 2001.

[11] D. S. H. Rosenthal and V. Reich, "Permanent web publishing," in *ATEC '00: Proceedings of the annual conference on USENIX Annual Technical Conference*. USENIX Association, 2000, pp. 129–140.

[12] L. Veiga and P. Ferreira, "Repweb: replicated web with referential integrity," in *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2003, pp. 1206–1211.

[13] T. A. Phelps and R. Wilensky, "Robust hyperlinks cost just five words each," EECS Department, University of California, Berkeley, Tech. Rep. UCB/CSD-00-1091, 2000. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2000/5442.html

[9]e.g., the W3C link checker, http://validator.w3.org/checklink

[10]http://www.changedetection.com/