# 6 IMPLEMENTATION AND EVALUATION

Up to the present, we have outlined the overall architecture and possibilities of an AHS, focusing on the development of Heuristics for adding quality to the process of building the domain model. In this chapter we first present an implementation of the strategy, demonstrating the resulting concepts and relations based on a small document collection. Next, we discuss and evaluate our work, concluding with some promising fields of future research.

## 6.1 Realising the domain model

Our strategy for conceptualization presumes that the domain documents are being parsed for important elements in order to build the domain model. Every term in these elements is a candidate concept. Guided by the Heuristics for concepts introduced in section 5.2.3, a conceptualizer module associates values with each candidate, leaving some with a higher score than others, and finally selects the most promising term as the concept. The same process apply at both the document and element level. The results from the parsing is passed on to another module for further analysis, which includes a merge of concepts and identification of relationship types.

Our HTML parser is written in `C` though any language would do for the task. The analysis is performed by `PROLOG`. Due to its properties as a language, `PROLOG` is very suitable for performing effective and powerful reasoning using only a few lines of programming code. The decision to use the combination of `C` and `PROLOG` is based on a wish for integrating the AHS routines into an existing agent based framework written in `C`. The agents at hand are `PROLOG` *machines* that can call `C` *routines* [*Thomassen99*]. Furthermore, `C` and `PHP` come well along, and suggestively, functions from `OpenGL` can easily be called from a `C` agent. All implementation specific details are attached in Appendix E - Implementation of conceptualizer.

### 6.1.1 A walk through the conceptualization processes

Before parsing, all hyper references in the domain are identified and stored in a file, and the values of the Heuristics for finding concepts are initialised. All the

documents in the domain are listed in a file of document names, one name for each line. The process of analysing the domain is therefore controlled by walking through the file line by line while opening the next document and analysing it. Since stemming is used throughout the analysis, the DSL is also stemmed to be on the safe side. Thereafter, the first document is parsed followed by all its elements classified as important, then the second document and its important elements, and so forth. Unless otherwise stated, from now on we refer to both documents and important elements as *knowledge sources*.

According to one of the principles of HCI, it is much easier to agree or disagree with some proposal than coming up with something new from scratch. For instance, a DSL could be constructed semi-automatically by proposing the highest weighted terms from each document, then allowing the author to edit and correct the list. The construction should be rather simple since it can basically be implemented as a text file of words regarded as being important.

The Heuristics for finding candidates rely on several methods for operating on the knowledge sources in order to incrementally adjust the values of each candidate. All sets of temporal or longer term results are written to text files with one term for each line, allowing for easy manipulation of appropriate sets in the combination of an ADT (abstract data type) List. Regularly used functions include list operations for comparison, searching for specific terms, sorting, counting occurrences of the elements, reading a file to its internal list representation, and removing duplicate members. The step of lexical analysis and removing stopwords has an expensive cost and could account for as much as 50% of the computational expense of compilation [*Frakes+92*]. An extremely efficient implementation is to remove stopwords as part of the lexical analysis, that is, functions for lexical analysis and stopword removal are intertwined through the use of a DFA[1]. All terms surviving this process are given values by counting term frequencies, according to Heuristic HC-2, and membership in the DSL is checked for (HC-1), adding the associated value to members. Similarly, emphasizers in the knowledge sources are identified (HC-3), while meta information is of great importance for documents only (HC-4).

The links play a dual role. To fresh up from the discussion in the previous chapter, terms in hyper reference elements from somewhere in the domain pointing to the document being analysed, are likely to be among its strongest candidate concepts. Since such hyper reference terms are strong candidates for the corresponding destination documents, they are not suitable as candidates for the document hosting the link element. In concordance with Heuristics HC-5 and HC-6, the file of all identified links in the domain guides the punishment and reward of candidates. As mentioned earlier, documents are parsed before its elements, as required by Heuristics HC-8 and HC-11, and occurrences of the selected document concept in the elements of a document are punished by adding negative values.

As the parsing makes progress several things happen at different levels. Of importance, duplicate combinations of the tuple <tagName, conceptName> are not

1. A Deterministic Finite Automaton (DFA) object represents words in a network of interconnected characters, so that a whole word is found by following paths of linked characters.

allowed for elements nor for documents, so the system rejects such proposals and tries to select among the consecutive candidate concepts until successful. The sections of the documents are being retagged and the knowledge sources identified are stored in individual, uniquely named files, all together constituting a new *knowledge base*. As far as reporting concerns, all candidate concepts are written to individual log files, one log file for each document. Additionally, the concepts selected are added to another file in a form understandable by `PROLOG` in order to facilitate further analysis. As illustrated in Figure 6–1: the file "parsing_results.pro" has information on each knowledge source, like conceptual representation, the element type[1], an ID for later retrieval from the already established knowledge base, and a *list* of candidates with scores close to that of the concept selected. The latter attribute (also referred to as "the upper list of candidates") facilitates evaluation and adjustment of the proposed domain model as the system might suggest other candidates if the author is displeased with the proposed DM, and, as seen in the previous chapter, they will also play an important role when it comes to the identification of certain relationship types.

**parsing_results.pro**

```
% ------------------------------------------------------------
% The values associated with the Heuristics:
%               HC-1  (DSL)        = 50                    HC-2  (term frequency)  = TF
%               HC-3  (emphasizers) = 10                    HC-4  (meta information) = 50
%               HC-5a (headings)    = 20                    HC-5b (punish links)    = -1000
%               HC-6  (linked to)   = 50                    HC-8  (punish doc_con)  = -500
% Attributes: (concept, element type, location, candidate list)
% ------------------------------------------------------------

kw_source(definit, html, "file1.html", [autonom, ascript, descript, mean, term, approach] , "file1.html").
kw_source(agent, ol, "file1.html#elm1", [autonom, ascript, descript, mean, sometim, attribut] , "file1.html").
kw_source(agent, html, "file2.html", [characterist, attribut, reactiv, proactiv, model, adapt] , "file2.html").
kw_source(reactiv, ul, "file2.html#elm1", [proactiv, model, adapt, autonom, knowledg, collabor] , "file2.html").
kw_source(autonom, ol, "file2.html#elm3", [intellig, degre, machin, margin, top] , "file2.html").
kw_source(interfac, p, "file2.html#elm4", [interfac, collabor, nana, classif, result, type] , "file2.html").
kw_source(dictionari, p, "file2.html#elm5", [act, behalf] , "file2.html").
kw_source(softwar, html, "file3.html", [interfac, intellig, user, direct, manipul] , "file3.html").
kw_source(intellig, ol, "file3.html#elm1", [user, interfac, agent, cooper, simplifi, distribut] , "file3.html").
kw_source(agent, table, "file3.html#elm2", [interfac, user, proactiv, reactiv, task, search] , "file3.html").
kw_source(debat, html, "file4.html", [mae, user, interfac, ben, adapt, domain] , "file4.html").
kw_source(user, ul, "file4.html#elm1", [agent, interfac, adapt, intellig, model, futur] , "file4.html").
kw_source(agent, ul, "file4.html#elm2", [user, domain, interfac, focus, speech, difficult] , "file4.html").
kw_source(foley, html, "../341/foley.html", [norman, stag, seven] , "../341/foley.html").
```

**Figure 6–1: Information about the knowledge sources are written to a file in a format understandable by PROLOG. For the document concepts, the ID's appear as filenames only. For each element within a document, its ID is composed as follows: "filename" + "#elm" + "number", where the number is increased as new elements are being analysed. Note that the concepts are stemmed. Also worth noticing is the uniqueness of the combination concept + tag, which agrees to the previous discussion.**

Each element originates from a document. The observant reader might wonder why document names are listed as well. Even though this information is held by the third attribute, or ID, it facilitates the implementation of one of the Heuristics.

1. Just as element types are labeled with the corresponding tag names, documents are labeled as "html".

The reason is that PROLOG is weak on string manipulation, even for a simple task like extracting the file name from the ID.

## 6.1.2 Seaming up the domain model

From the file in Figure 6–1 we see that some conceptual descriptions coincide. Note that the temporary division of document concepts and element concepts (c.f. section 5.1.4) were useful for splitting up the domain knowledge into appropriate knowledge sources, and for discussing how to capture the conceptual hierarchy. For the final DM, however, this division is superfluous since the conceptual hierarchy will be reflected through the relations. Furthermore, the file "parser_results.pro" must be kept intact for the future adaptation phase, since it holds information on which knowledge sources each concept represents. Therefore, listing all the conceptual descriptions as PROLOG facts in a new file while removing duplicates, elegantly solves the first part of completing the DM. We want a PROLOG representation of the domain knowledge in terms of abstracted concepts and how they are related. The left part of Figure 6–2 shows the domain model as such a file. To the right, the DM it is visualised as a conceptual structure.

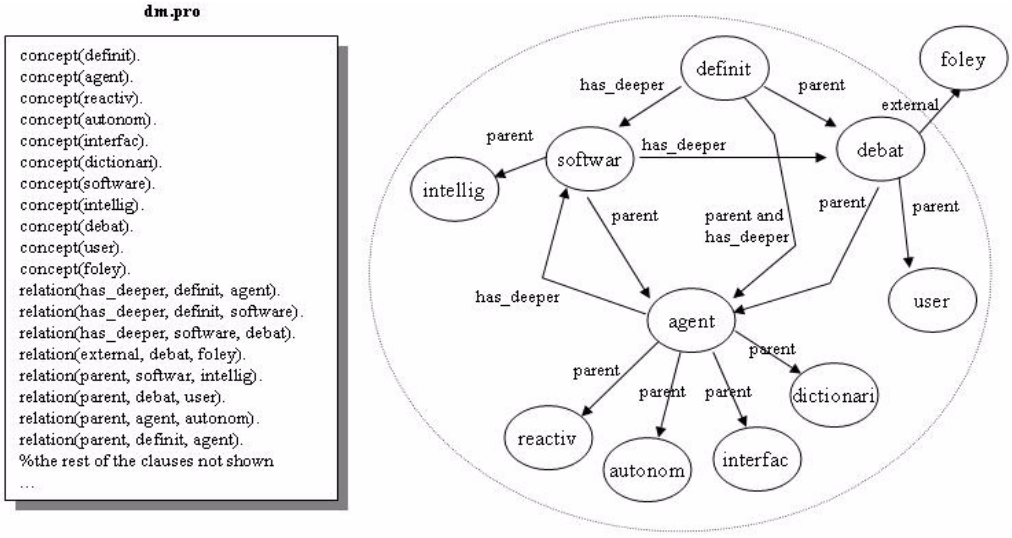


**Figure 6–2: The domain model represented in a PROLOG file.**

From the two illustrations shown by now, we see that one goal to be accomplished is to abstract the concepts only, from all the information held by the fact kw_source, where new facts should be written on the form

concept(ConceptName). This task can be implemented straightforward as illustrated by the rather simple PROLOG code below:

**Table 6–1: PROLOG code for collecting all concepts and writing these as new facts.**

```
%------------------------------------------------------------------------
% This code collects each concept from the kw_source fact,
% and writes the result as a new fact.

make_concept :-
     kw_source(ConceptName, _ , _ , _ , _ ) ,
     assert( concept(ConceptName) ) ,
     fail.
     % fail forces backtracking so that every combination is tried

make_concept.
     %when everything is tried and the above rule fails, this one succeeds
```

Heuristics HR-1 and HR-2 sought to find deeper explanation relationship types, doing so through the use of already identified hyper references. Remember that the initial step of conceptualization was to file all hyper references of the domain, which was then necessary to help the parser to point out concepts. Now this file becomes a useful source for the relational Heuristics HR-1 and HR-2. Figure 6–3 illustrates a small sample of links generated from the analysis of a domain.

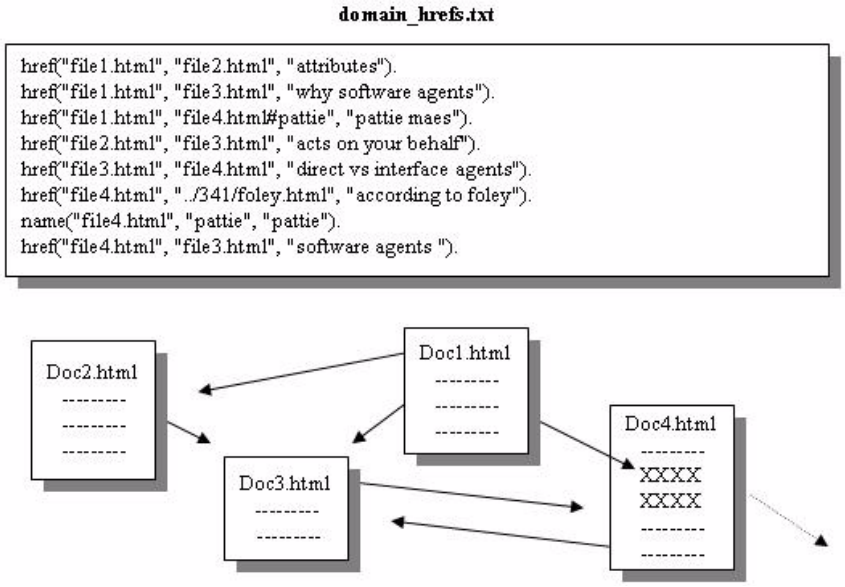


**Figure 6–3: The file based on linkage between the original documents. Note that one link in Doc1.html points to a specific section of Doc4.html, which in turn has a link directed to somewhere outside the domain (namely to a document debating ("foley"). The destination in Doc4.html that is named "pattie" leads to the identification of a name-clause.**

As mentioned, the domain was parsed based on a listing of all document filenames. The domain is useful when identifying relations, so this file is transformed to a PROLOG fact using a list. That is, for a domain with e.g. the four files "file1.html", "file2.html", "file3.html" and "file4.html", the corresponding PROLOG fact takes on the form:

```
domain([“file1.html”,“file2.html”,“file3.html” ,“file4.html”]).
```

a task which can be accomplished e.g. by the C-routines. Furthermore, by combining facts about hyper references with the facts on the knowledge sources, PROLOG routines can find out which sections are related and link the corresponding concepts. The built-in predicates `assert` and `retract` makes it possible to update the program with new clauses or delete clauses during program execution, another advantage with the PROLOG language. In other words, as the program precedes, it can modify itself. This convenience is used thoroughly in the implementation of the Heuristics, and was also used when making concepts out of the `kw_source` fact (c.f. Table 6–1). The code for the first two Heuristics are illustrated below.

**Table 6–2: Heuristics HR-1 and HR-2 expressed in PROLOG.**

```
%-------------------------------------------------------------------------
% HR-1: Hyper references between two knowledge sources somewhere within
% the domain are also relations of type has_deeper_explanation between the
% corresponding concepts.

hr1 :-
     href(From, To, _ ) ,
     kw_source(ConceptA, _ , From, _ , _ ) ,
     kw_source(ConceptB, _ , To, _ , _ ) ,
     assert( relation(has_deeper_explanation, ConceptA, ConceptB) ) ,
     fail.    % forces backtracking

hr1. % ensure success

%-------------------------------------------------------------------------
% member is recursively defined and finds out whether an X is in a list.

member(X, [X | Tail]).
member(X, [Head | Tail]) :-
     member(X, Tail).

%-------------------------------------------------------------------------
% The domain is constrained to a list of valid filenames, here exemplified
% with four files only. The rule outsideDomain checks for membership in
% the domain list.

domain( [“file1.html”, “file2.html”, “file3.html”, “file4.html”] ).

outsideDomain(F) :-
     domain(D) ,
     not member(F, D).

%-------------------------------------------------------------------------
% HR-2: External links with destinations outside of the domain
% are slightly different from internal ones, and the corresponding
% relations between such knowledge sources should be labelled as external.

hr2 :-
     href(From, To, _ ) ,
     outsideDomain(To) ,
     kw_source(ConceptA, _ , From, _ , _ ) ,
     kw_source(ConceptB, _ , To, _ , _ ) ,
     assert( relation(external, ConceptA, ConceptB) ),
     fail.    % forces backtracking

hr2. % ensure success
```

As an example, the task for HR-1 is to find concepts that are linked based on document linkage and we therefore start with establishing the source (the variable `From`) and the destination (the variable `To`) of each link by searching the `href` facts listed in Figure 6–3. Then we need the document concept from the source of the link and the document concept from the link destination, so we pick only these from the `kw_source` facts listed in Figure 6–1, elegantly ignoring the element types and the list of candidates using the `PROLOG` understandable underscore character as argument. Note that due to the form of the arguments of `href` and the form of the ID argument from `kw_source`, we don't have to account for element types, i.e. only document concepts will be selected. The set of Heuristic for the conceptual hierarchy can also be easily implemented in `PROLOG`. Before finishing the rule for HR-3, we note that when the filenames equal for two concepts, they denote the same document, and the element type "html" denotes the document concept.

**Table 6–3: Implementing the Heuristics for finding parent and synonym relationship types.**

```
%-------------------------------------------------------------------------
% HR-3: All element concepts found within a document are children
% of the document concept.

hr3 :-
     kw_source(DocumentConcept, html, _ , _ , File) ,
     kw_source(ElementConcept, _ , _ , _ , File) ,
     not DocumentConcept = ElementConcept ,
     assert( relation(parent, DocumentConcept, ElementConcept) ) ,
     fail.    % forces backtracking

hr3. % ensure success

%-------------------------------------------------------------------------
% HR-4: There is a parent relation if a member from a set of
% candidate concepts equals an already found concept.

hr4 :-
     kw_source(ConceptA, _ , _ , CandidateList, _ ) ,
     kw_source(ConceptB, _ , _ , _ , _ ) ,
     not ConceptA = ConceptB ,   %  the two concepts should not be equal
     member(ConceptB, CandidateList) ,
     not relation(parent, ConceptA, ConceptB) , %only consider once
     assert( relation(parent, ConceptA, ConceptB) ) ,
     fail.    % forces backtracking

hr4. % ensure success

%-------------------------------------------------------------------------
% HR-5: If two concepts have some joint members from their
% set of candidates, the concepts are synonymous.

commonMember(List1, List2) :-
     member(Term, List1) ,
     member(Term, List2).

hr5 :-
     kw_source(ConceptA, _ , _ , CandidateListA, _ ) ,
     kw_source(ConceptB, _ , _ , CandidateListB, _ ) ,
     commonMember(CandidateListA, CandidateListB) ,
     not ConceptA = ConceptB ,
     not relation(synonym, ConceptA, ConceptB) , %only consider once
     assert( relation(synonym, ConceptA, ConceptB) ) ,
     fail.    % forces backtracking

hr5. % ensure success
```

The last two Heuristics will aim at finding prerequisite relations. Similar to converting the file with the domain document listing, we also assume that the C-routines produces a PROLOG understandable version of the DSL. In order to make Heuristic HR-6 work, we need information on the terms that is not in the upper list of candidate concepts, in other words those terms that are not present in the last argument of clauses named kw_source. Remember from the previous discussion that such a list exists for each element, namely in the log file for each document. Assuming that the C-routines also convert this list to PROLOG facts on the form lowerList(ID, ListOfLowerScoreTerms), we are set to implement this Heuristic.

**Table 6–4: "Prologging" the first prerequisite relationship type.**

```
%------------------------------------------------------------------------
% Lower score terms are listed as PROLOG facts, here illustrated with
% three examples. Note that the lists in lowerList and kw_source
% together constitute all the terms surviving lexical analysis.

lowerList("file4.html", [speech, futur, direct, manipu, design, system]).
lowerList("file4.html#elm1", [design, system, speech, memori, direct]).
lowerList("file4.html#elm2", [softwar, disagr, mainli, due, ben]).

inLowerList(Term, Location) :-
     lowerList(Location, Z) ,
     member(Term, Z).

%------------------------------------------------------------------------
% The domain specific list contains terms that are regarded important,
% here exemplified with seven terms only.

domainSpecificList( [hci, adapt, iui, user, model, domain, intellig] ).

inDsl(Term) :-
     domainSpecificList(DSL) ,
     member(Term, DSL).

%------------------------------------------------------------------------
% HR-6: A term in a knowledge source KS_1 which is neither selected as
% concept nor in the upper list of candidates, yet a member of the DSL
% and chosen as concept for another knowledge source KS_2,
% is prerequisite to the concept of the first knowledge source KS_1.

hr6 :-
     inLowerList(Term, Location) ,
     inDsl(Term) ,
     kw_source(Concept , _ , Location, _ , _ ) ,
     concept(Term) ,
         %does the concept of the source equals the term?
     assert( relation(prerequisite, Term, Concept) ) ,
         %if so, then make a relation between the two.
     fail.    % forces backtracking

hr6. % ensure success
```

The PROLOG code of Heuristic HR-6 seems more complex than those of the previous rules. Will it generate the prerequisite relation correctly? Refresh on the situation of Figure 5–9 (which illustrated Heuristic HR-6) where a regular term *klm*, which also occurred in the DSL, was only briefly mentioned in a document. Since *klm* already had been identified as a concept somewhere else, and *goms* was voted as concept of the document in which *klm* lived as a regular term only, we claimed *klm* to be prerequisite to *goms*. In order to explain the implementation, we

illustrate by including only the facts necessary. Note that the content of Table 6–5 corresponds to the *klm/goms* situation, where the two clauses `concept(goms).` and `concept(klm).` were obtained by calling the previously stated rule `make_concept`.

**Table 6–5: A small portion of the knowledge on two documents as seen from PROLOG.**

```
kw_source(goms, ul, "file5.html", [operator, mhp, analysis], "file5.html").
kw_source(klm, p, "file7.html", [keystroke, button, time], "file7.html").

lowerList("file5.html", [mouse, klm, movement]).

domainSpecificList( [hci, foley, norman, goms, klm, mhp, gestalt] ).

concept(goms).
concept(klm).
```

To start, we ask `hr6.` in order to identify the prerequisite relations which conditions to the Heuristic. The first two goals tries to satisfy membership in the `lowerList` and the `domainSpecificList`, using the rules from Table 6–4. For successful bindings to the variable `Term` (in our case the second member of the lower list, that is *klm*), the `Location` variable is further used in order to identify the concept of the knowledge source. Next, we must ask whether the term found is also a concept, simply by posing the question `concept(Term).` Since there is such a clause for the present term, namely `concept(klm).` the question succeeds, and a prerequisite relation can be inserted as part of the knowledge database. Finally, in order to validate, we ask `PROLOG` the question `relation(prerequisite, A, B).` to check whether any relations were found. The result is that the variables `A` and `B` are bound to *klm* and *goms* respectively, in other words `PROLOG` found the newly inserted fact `relation(prerequisite, klm, goms).`

The last Heuristic HR-7 assumes there is a path between two related concepts. Thus, its solution involves the traversal of several paths. We therefore start by defining the code for how to find a path, and doing so requires the use of *edges*, (a path is made of edges between the nodes). Note that we already have the edges established from the `href` facts. If a path exists along with an already discovered relation, the procedure `addToSet` starts to extend the database with new prerequisite relations. Finally, the original relation is removed. The code for all this

is summarised below and will not be further explained. Again, notice the compact form due to the properties of the PROLOG language.

**Table 6–6: Paths are the basis for finding more prerequisites.**

```
%-------------------------------------------------------------------------
% There is a path between two nodes if there is an edge from the first
% node to an intermediate node, which again has a path to the destination
% node. This code also adds the nodes to the head of a list.

path(From, To, [To] ) :-
     href(From, To, _ ).

path(From, To, [Intermediate|Tail] ) :-
     href(From, Intermediate, _ ), %adds to head of list
     path(Intermediate, To, Tail).

%-------------------------------------------------------------------------
% This rule asserts prerequisite relations from the intermediate
% members of a path to the last node

addToSet([Head|Tail], ToConcept) :-
     kw_source(FromConcept, _ , Head, _ , _ ) ,
     not FromConcept = ToConcept ,  %write relation if the concepts differ
     assert( relation(prerequisite, FromConcept, ToConcept) ) ,
     addToSet(Tail, ToConcept) ,  %move down the path

addToSet([], _). %trivial case

%-------------------------------------------------------------------------
% HR-7: In the presence of a relation between two concepts, together with
% an unique, explicitly stated path of documents connecting the two
% corresponding knowledge sources, there is a set of prerequisite
% relations between the concepts of each document (but the first one),
% and the destination of the path.
% Note that the procedure removes the original relationship type if
% some prerequisites are found

hr7 :-
     relation(Type, ConA, ConB) , %if any relation exists
     kw_source(ConA , _ , From, _ , _ ) , %we go get the
     kw_source(ConB , _ , To, _ ,  _ ) ,   %destinations
     path(From, To, Visited) , %is there an explicitly stated path?
     addToSet(Visited, ConB) , %add all relations along this path
     retract( relation(Type, ConA, ConB) ) , %remove original relation
     fail.    % forces backtracking

hr7. % ensure success
```

By now, we have seen that all concepts and relations can be found using PROLOG. Together, they provide the basis for completing the domain model. Once this information is merged into one file (e.g. “dm.pro” used in Figure 6–2), this file essentially constitute the domain model, that is, one single file can represent the domain knowledge. The Heuristics work independently of each other, and hence there is also a chance that some concepts are linked with more than one relation. In particular, some Heuristics produce bi-directional relations. For instance, Heuristic HR-1 produces the relations

```
relation(has_deeper_explanation, software, debat).
relation(has_deeper_explanation, debat, software).
```

In order to fix this problem, one of the superfluous clauses can be retracted from the PROLOG database. The rule removeBidirectional in Table 6–7 shows how.

**Table 6–7: Some clauses to clean up bi-directional relations**

```
removeBidirectional :-
     relation(Type, ConceptA, ConceptB) ,
     relation(Type, ConceptB, ConceptA) ,
     retract( relation(Type, ConceptB, ConceptA) ) ,
          %removes the latter bidirectional relation
     fail.    % forces backtracking

removeBidirectional. % ensure success
```

A well known principle of design is that human beings find it easier to overrule errors from a proposal than coming up with one themselves from scratch. As noted, someone skilled in the domain should revise the DM in order to secure that only appropriate concepts and relations remain. We suggest a graphical visualization in the form of a conceptual network, though other presentational forms are possible. Note that the number of concepts might get out of hand. Therefore the system should provide means for "switching off" some concepts and hence leaving only part of DM visible. Such a demand can be fulfilled simply through hiding the subordinate concepts of the parent relations, that is zooming in or out until the desired level of detail is achieved (when using a graphical interface). In a similar fashion, if the author prefers a pruned network, the system should offer to permanently delete all concepts and relations not visible. Moreover, for the human reviser, the file "parsing_results.pro" proves to be useful in many ways. As an example, its list attribute, whose members are ranked by value, will facilitate tasks like renaming since new concepts can be proposed as substitutes for those incorrectly suggested (by the system) in the first place.

# 6.2 Possibilities for the AHS

In the following we briefly sketch an example of how a simple user model and adaptation model might look, thus substantiating the choice of a PROLOG based implementation. We focus on the representation of the conceptual knowledge and generic user information and preferences.

## 6.2.1 Picturing a simple user model

As outlined in the previous chapter, UM must hold information on the user's level of conceptual knowledge. A simple scheme is to save information on which concepts the user already has knowledge and at what level. The user model should be incrementally built so that it at any time can provide the adaptive hypertext system with information on what the user knows. Since one concept can represent many different knowledge sources, it is important that the user model also records which knowledge sources have been presented for each concept. Continuing with

PROLOG notation, we use a fact `knowledge` which has attributes on the *concept*, along with a *list* of knowledge sources that have been shown for each concept, and finally a *state* indicating which knowledge level the user has on each concept. For instance, the state of the concept "debat" is set to `deeper` since the user has accessed the external resource conceptualised as "foley" (c.f Figure 6–2). Since we regard UM as an overlay model of DM, it seems natural to also include the corresponding relations as new concepts are learned, that is, relations to neighbour concepts. Why embed relations in the user model? There are at least one reason. When a concept is learned and its neighbours are not, the knowledge gap should be bridged, and if the relations are kept in the domain model only, the implementation of the adaptation rules becomes cumbersome. Figure 6–4 shows how the file

**um.pro**

```
% ------------------------------------------------------------------
% User model of user "Lady Ada Hyper", female, born 1815
% ------------------------------------------------------------------
learning_style([graphical, philosophical]).
capability_level(intermediate).

knowledge(agent, ["file1.html#elm1", "file4.html#elm2"], incomplete).
knowledge(user, ["file4.html#elm1"], complete).
knowledge(debat, ["file4.html"], deeper).
knowledge(foley, ["www.foley.html"], complete).
relation(parent, agent, reactiv).
relation(parent, agent, autonom).
relation(parent, agent, interfac).
relation(parent, agent, dictionary).
relation(parent, debat, user).
relation(parent, debat, agent).
relation(external, debat, foley).
...
```

**Figure 6–4: A simple user model representing user knowledge at various levels. Note that only relations to neighbour concepts of the concepts already learned, are included in UM.**

"um.pro" represents a simple user model[1]. Note that due to the uniqueness of the tuple <tagName, conceptName>, the information in the tag list can be either in terms of the ID or as the tag name (we prefer using the ID, in case the author, despite warnings from the system, overrules the recommendations and assigns the same concept for two equal elements).

A user model should present information according to the needs and preferences of each user with regard to presentational form and learning abilities. Thus, by including some generic information on the learning styles and skills of each user in UM, the system is not constrained to a selection among the concepts not yet learned. Along with unknown concepts, also the preferences of each user can influence the adaptive commitments, so that presentational form and degree of difficulty can be further varied.

1. Possible extensions include e.g. the time spent on each node and interaction history.

From the clauses

```
learning_style( [graphical, philosophical] ).
capability_level(intermediate).
```

we see that the user “Lady Ada Hyper” prefers the system to generate visualizations if possible, a task that can easily be accomplished simply by asking for images and tables of the concept being debated. Remember that this information can be found from the attribute *element-type* in the `kw_source` facts. Likewise, the system has somehow inferred the capability of the user to be `intermediate`. It would therefore e.g. present easier concepts before the more difficult ones, but still challenge the user on more difficult subjects, thus aiming at improving the user knowledge without exaggerating.

## 6.2.2 Adaptation rules

We believe that the implementation of DM and UM in terms of facts has made a powerful foundation for the adaptive phase, and opened for various and flexible adaptive presentations, which can be fairly quickly generated by means of `PROLOG` rules. According to the introductory part on adaptive systems in section 4.5 “Planning an adaptive hypertext system”, page 29, the actions or tasks of an adaptive hypertext system are controlled by the adaptive engine operating on an adaptation model. In accordance with our work so far, a task would lead the adaptive engine to execute a query, and the rules in the adaptation model would ensure that the correct action can be executed. Three independent tasks to illustrate example rules are proposed in Table 6–8. The tasks will be explained in the following. Note that the clauses of the last rule are explained in detail after discussing the first two rules.

**Table 6–8: Examples of how easily powerful adaptation rules can be written in PROLOG.**

| Description of task | Example of AE call | Rule that fires in AM |
|---|---|---|
| 1. Provide graphical information about a concept to the user. | `?- show(agent,img).` | `show(Concept, ElmType) :-`<br>`  kw_source(Concept, ElmType,`<br>`  FileID, _ , _ ) ,`<br>`  display(FileID).` |
| 2. Based on a concept in DM, visualise a sub-network of relations and neighbour concepts not yet visited. | `?- visualise(user).` | `visualise(Concept) :-`<br>`  concept(Concept) ,`<br>`  relation(Type, Concept, Neighbour) ,`<br>`  draw(Concept, Neighbour, Type) ,`<br>`  fail. %get all neighbours`<br>`visualise(Concept). %ensure success` |
| 3. Propose new concepts to be presented to the user based on the gap close to a given concept in the user model | `?- next(softwar).` | `next(Concept) :-`<br>`  getNeighboursDM(Concept, DMList) ,`<br>`  userKnows(Concept, UMList) ,`<br>`  findGap(DMList, UMList, GapList).` |

The first task listed concerns to embed graphical information on a concept in the adaptive document being presented to the user. It assumes the parser found exactly that concept for an image element during parsing the documents. If so, the image to be presented is contained as HTML code within the knowledge base in a small file whose reference is kept in the third attribute of `kw_source`, so the solution is simple: Query the Adaptive Engine to display the correct `fileID` if the associated concept and the correct type proves to be true for a knowledge source. Note that we assume that the rule `display(FileID)` exists without focusing on its implementation. Somehow, it should return the `fileID` to the system for generation of the adaptive document to be presented. One way to do this is for display to write include-sentences to a file "includes.php", which e.g. a `PHP` script traverses and uses further for generation of the final documents. Note this is very simple for `PHP` since it can tailor many different documents by simply including their filenames.

The second task provides a visualization of the nearby network of a concept upon request. Note that whereas the information is provided by simple `PROLOG` clauses, the actual routines for drawing might be more complex and typically performed by a routine written in another programming language, typically `OpenGL`. As noted, one reason for choosing `PROLOG` and `C` is that the two can communicate within an already existing agent based framework, and `OpenGL` routines can be called from `C`.

Before considering the third task, we recall that the goal of adaptation is to satisfy the user with respect to knowledge, and in order to do so, a plan for which concepts to present next and how to present them, is needed. There are various strategies for filling up the user model, each leading to different adaptation rules. One strategy is to search the domain model in a breadth first manner and introduce all concepts briefly before focusing on increasing the user understanding of each concept. Cycling through the domain model in order to cover up incomplete user knowledge would gradually perfect the user model. More likely than benefiting from many walk-throughs, the user could experience all the jumping and revisiting to be annoyingly inefficient and loose track of the knowledge to be learned. The opposite strategy follows a depth first like search through the domain, aiming at fully describing each concept in one operation. A problem with finishing off each concept before regarding new ones is that the user could miss sight of the greater picture.

Another strategy for guiding the selection process is based on the relations identified so far. Using relations when searching for new pieces of knowledge includes several subordinate tasks to be accomplished. Remember that the gap of knowledge in the user model consists of *unrelated* areas in the user model with corresponding *related* areas in the domain model. The task then comes to bridge the gap of knowledge using the relations of DM. Note from the below illustration that the concept `softwar` in DM is related to the concepts `intellig`, `agent`, `definit` and `debat`, whereas the user has learned the concepts `software` and `agent` only, as indicated with empty circles. In other words, there is a gap close to the concept `software` which should be bridged, and the concepts `intellig`

and `definit` constitute the building blocks to the piece of engineering work at hand.

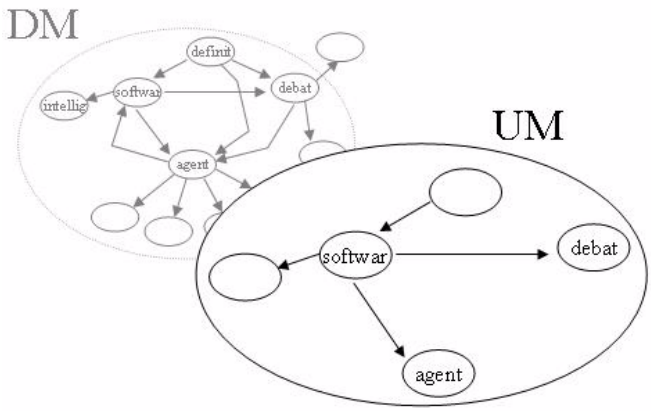


**Figure 6–5: The gap in the user model.**

The strategy is to let lists represent both UM and DM, and show all members from UM not also in DM. The first step is to identify a list of neighbours from DM close to a concept, doing so by calling the rule `getNeighbours(Concept, DMList).` and now `DMList` holds all nodes close to the concept. Similarily, a list of user model neighbours to the concept must be found. The next step of the strategy is to use the rule `findGap(DMList, UMList, GapList).` to subtract the concepts of the user model from the members of the domain model. This operation is basically a process of deleting every member of `UMList` from `DMList`, resulting in another list `GapList` holding the concepts to be displayed to the user. Using our example, `DMList` initially looks like this:

```
[agent, intellig, definit, debat]
```

and the `UMList` contains

```
[agent, debat]
```

so `GapList` would after the `findgap` rule is run, yield

```
[intellig, definit]
```

Note that the system should also validate whether other concepts are required to be known before displaying the members of `GapList`, placing possible prerequisites along with the gap concepts found in a final list, which should then be traversed

and displayed to the user. Table 6–9 sums up the rules needed for bridging gaps, but does not embed the rules needed to validate prerequisites.

**Table 6–9: Examples of adaptation rules written in PROLOG.**

```
%-------------------------------------------------------------------------
% The rule next(Concept) finds a gap and builds a bridge, that is a list
% of concepts that can be visited next. Note that this routine does not
% validate the prerequisites of the concepts in the gap.

deletepossible(Item, _ , [] ).
deletepossible(Item, [Item|Tail] , Tail).
deletepossible(Item, [Y|Tail] , [Y|Tail2] ) :-
     deletepossible(Item, Tail, Tail2).

insert(X, List, BiggerList) :-    %insert is the inverse of delete
     delete(X, BiggerList, List).

findGap( [] , R, R).  %trivial case
findGap( DMList, [Head|Tail], GapList):-
     deletepossible(Head, DMList, IntermediateList) ,
             %deletes common members
     findGap(Tail, IntermediateList, GapList).

all(Concept, ListIn, ListOut) :-   %go get all neighbours
     relation( _ , Concept, Next) ,
     not member(Concept, ListIn) ,
     all(Concept, [Next|ListIn], ListOut).
all(C, List, List). %trivial case

getNeighbour(Concept, DMList) :-  %get neigbours to a concept in DM
     all(Concept, [], DMList).

userKnows(Concept, UMList) :-    %get concepts known to the user.
     all2(Concept, [], UMList).  %almost equal to all, except that
                                 %relation in all is the DM relation,
                                 %and relation i all2 is UM relation.

next(Concept) :-
     getNeighboursDM(Concept, DMList) ,   %list all neighbours from DM
     userKnows(Concept, UMList) ,  % build list of neighb. known to user
     findGap(DMList, UMList, GapList) ,  %subtract UM from DM
```

# 6.3 Evaluation

The goal of the conceptualizer module was to abstract the contents of documents and the sections as good as possible. Rules were then used to complete DM with relations between the appropriate concepts. Fine, but are the results reliable? How can we justify the actions taken? What if the structure of a collection of documents deviate from the assumptions of some Heuristics? This section discusses the influence of the commitments made.

## 6.3.1 Optimizing and justifying the performance

In section 5.2.4 we suggested that it is possible to experiment with the total outcome of the conceptualization process by varying the values associated with each Heuristic. The candidate concepts are terms, where the one with the highest

value after all Heuristics are performed, is selected as concept. For every term $\mathbf{t}_i$ an Heuristic can either apply or not, indicated in Table 6–10 with “Yes” and “No”, respectively. The columns indicates which of the Heuristics do fire for each term. Based on some few possible combinations, this section tries to draw some conclusions and apply suitable values for an optimal combination of the values donated from each Heuristic. Note that HC-5 is implemented in two portions in order to account for its dual nature while Heuristics HC-7, HC-9, HC-10 and HC-11 are left out since they mainly deal with issues like the level of abstraction, sequence of analysis etc.

**Table 6–10: The outcome of a term is determined by the total score, which is due to the effect from each Heuristic. Some of the possible combinations are listed in the columns.**

| Heuristic | $\mathbf{t}_1$ | $\mathbf{t}_2$ | $\mathbf{t}_3$ | $\mathbf{t}_4$ |
|---|---|---|---|---|
| **HC-1** (Domain specific list) | No | Yes | No | Yes |
| **HC-2** (Term frequency) | Yes | Yes | Yes | Yes |
| **HC-3** (Emphasizers) | No | No | No | Yes |
| **HC-4** (Meta and title) | No | No | No | No |
| **HC-5a** (Occurrences in different elements) | No | No | No | No |
| **HC-5b** (Punish outgoing links) | No | Yes | Yes | No |
| **HC-6** (Incoming links pointing here?) | No | No | Yes | Yes |
| **HC-8** (Punish document concept) | No | No | No | Yes |

For instance, the term $\mathbf{t}_1$ does not appear in the DSL, nor in any important element, so the only Heuristic that strikes, is Heuristic HC-2, as shown in its column. We make several notes from the above combinations:

1. All candidates have a term frequency (HC-2), but the number will vary from one and up. In the worst case, no Heuristics but the second one proves correct, as for $\mathbf{t}_1$.
2. A term might well occur both in the DSL and in outgoing links, as illustrated with $\mathbf{t}_2$.
3. There are terms like $\mathbf{t}_3$ that occur both in outgoing and incoming links (HC-5b and HC-6) at the same time. With incoming links, we think of links from elsewhere pointing to a specific section hosting the term.
4. For an element, all Heuristics but the last one suggests assigning positive values to the candidate $\mathbf{t}_4$.

Instead of testing empirically which combination of values yield the better results, we try to analyse the points observed above. From the first we conclude that the values must agree with the term frequencies so that they can affect the total outcome. The second point reveals a problem. Outgoing links are subject for punishment (negative value) whereas DSL members obviously should yield a high, positive value. Which Heuristic should veto? Since the DSL apply for all documents, we infer that HC-5b should overrule HC-1. The third case concerns instances where a candidate is regarded likely to be the concept of another document and the present one at the same time, due to membership in hyper

references. In order to resolve this conflict, we note that since links are only assumed, not for certain, to be descriptive of their targets, the sum of the two should neutralize each other. Finally, even though the first Heuristics seem to assign plenty of positive values for an element, the same candidate is already chosen as the document concept. It should therefore not be selected again as concept for any element in the document. It is necessary for HC-8 to have a negative value able to beat the sum of the positively minded Heuristics HC-1, HC-3 and HC-6, also accounting for relatively high term frequencies.

Almost like solving a mathematical equation, appropriate values can be set. Note a little trial and error will do in order to find values that satisfy all of the above demands. As one solution, Figure 6–6 reveals how the results (the `kw_source` facts) of an optimal set of Heuristics (the header section) differ from the setup used for the other examples of this chapter, thus outlining the importance of committing to appropriate values. Note that this is favourable rather than a limitation for the system since it allows for a variation of proposed models made by the system. Additionally the author can decide which Heuristics to apply or omit, the latter being fruitful in case a collection of documents uses some tags for other purposes than outlined in this thesis.

**parsing_results.pro**

```
% ------------------------------------------------------------------------------------------------------------
% The values associated with the Heuristics:
%               HC-1  (DSL)       = 100                    HC-2  (term frequency)   = TF
%               HC-3  (emphasizers) = 10                   HC-4  (meta information) = 30
%               HC-5a (headings)   = 20                    HC-5b (punish links)     = -200
%               HC-6  (linked to)  = 200                   HC-8  (punish doc_con)   = -400
% Attributes: (concept, element type, location, candidate list)
% ------------------------------------------------------------------------------------------------------------

kw_source(autonom, html, "file1.html", [definit, ascript, descript, mean, term, approach], "file1.html").
kw_source(agent, ol, "file1.html#elm1", [ascript, descript, mean, sometim, attribut, person], "file1.html").
kw_source(attribut, html, "file2.html", [agent, reactiv, proactiv, model, adapt, autonom], "file2.html").
kw_source(reactiv, ul, "file2.html#elm1", [proactiv, model, adapt, autonom, knowledg, collabor], "file2.html").
kw_source(autonom, ol, "file2.html#elm3", [intellig, degre, machin, margin, top], "file2.html").
kw_source(agent, p, "file2.html#elm4", [interfac, collabor, nana, classif, result, type], "file2.html").
kw_source(dictionari, p, "file2.html#elm5", [act, behalf], "file2.html").
kw_source(agent, html, "file3.html", [softwar, interfac, intellig, user, proactiv, reactiv], "file3.html").
kw_source(intellig, ol, "file3.html#elm1", [user, interfac, cooper, simplifi, distribut, comput], "file3.html").
kw_source(interfac, table, "file3.html#elm2", [user, proactiv, reactiv, task, search, action], "file3.html").
kw_source(debat, html, "file4.html", [mae, user, interfac, adapt, domain, intellig], "file4.html").
kw_source(user, ul, "file4.html#elm1", [agent, interfac, adapt, intellig, model, futur], "file4.html").
kw_source(agent, ul, "file4.html#elm2", [user, domain, interfac, focus, speech, difficult], "file4.html").
kw_source(foley, html, "../341/foley.html", [norman, stag, seven], "../341/foley.html").
```

**Figure 6–6: The values should be optimal in accordance with the observations listed in Table 6–10. Note that these values result in different knowledge sources than the values used in Figure 6–1. Hence, other concepts and relations are produced for the final domain model.**

By leaving Heuristic HC-1 with a value of zero, i.e. neglecting the DSL, the final DM would again turn out different. Note, however, that in the absence of a DSL, both the conceptualization and the identification of the prerequisite relationship type would suffer. In a similar fashion, we may expect the granularity of the DSL regarding content to affect the grand total. Therefore both the DSL and the values

of the Heuristics can be regarded as tools for which we can experiment on the final outcome.

### 6.3.2 An ocean of relations

The code for the relations can very easily be tested using any `PROLOG` environment. For the four sample documents listed in appendix C (which is also the basis for most of the screen-shots from this chapter), the results are clear: quite a lot of relations were found, especially due to the parent- and synonym contributions:

- 4 instances of the *has_deeper_explanation* relationship type were found.
- 1 instance of the *external* relationship type was found.
- 21 parent relations were found, some duplicates and bi-directional occurrences were removed.
- 23 synonym relations were found.
- 0 prerequisite relations were found.

Note that even though zero prerequisites were found for the four documents, we believe that the rules of Heuristics HR-6 and HR-7 are promising. During testing, the rules correctly caught the prerequisite relationship as outlined in Figure 5–9 and Figure 5–10. Anyway, the figures are scary. Altogether 69 relations were found. In comparison, only 11 unique concepts were found, which means that on the average each concept has more than 6 relations. Of course these numbers may differ on larger collections. Another important factor is the construction of the sample documents. More profound testing and possible revision of the rules is needed, but beyond the scope of this thesis. Questionable, therefore, how else can we secure that the system performs tolerably well given the proposed framework? Remember that the author has been quite silent all the way through, except when creating the DSL and pushing some buttons in order to start the processes of building the domain model. Let us therefore close up with putting a load on the author.

### 6.3.3 Document concepts vs. element concepts

An early commitment as part of the process towards a domain model, was to separate document concepts from element concepts. The decision was based on the observation that documents have a context superior to its sections and other subordinate documents. Indeed, the separation gave birth to the three different sets of Heuristics for conceptualization, and later on proved convenient for the identification of the parent relationship type.

As a side-effect, the separation also produced knowledge sources of different granularity. First, the original documents were preserved and conceptualised with the second attribute set to `html`, as exemplified in the `kw_source` fact

```
kw_source(softwar, html, "file3.html",
         [interfac, intellig, user, direct, manipul], "file3.html").
```

but also the elements of each document were written to individual smaller files, like in

```
kw_source(dictionari, p, "file2.html#elm5", [act, behalf] , "file2.html").
```

From the latter example we see that both the `ID` and the element type `p` indicates that this knowledge source originally was part of another document.

Is it useful to preserve both knowledge sources in the new knowledge base? One goal of adaptation is to tailor new documents that differ from the original ones, if more suitable for the user. The issue on document concepts versus element concepts are important as they are an essential part of the foundation for the work. We believed the system would *benefit* from this separation, but another view is that the resulting domain model actually will be limited *due to* the choice. The separation could in the worst case preserve the original hierarchy and structure so much that the resulting domain model will not capture anything but the original link structure. On the other hand, as indicated in Figure 6–7 there are two chooses for how to view the present knowledge base:

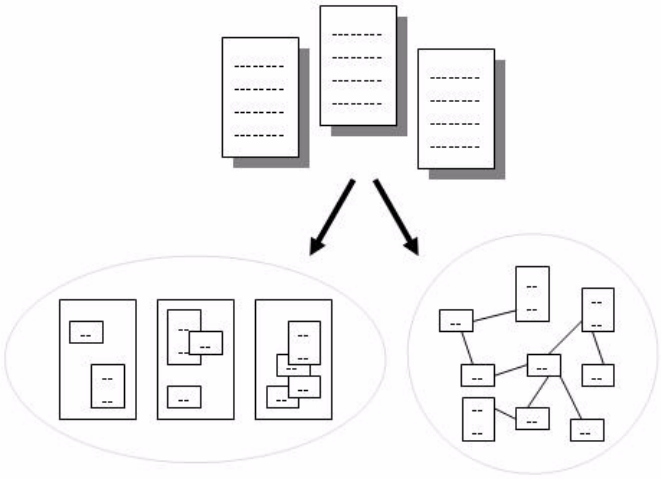

**Figure 6–7: The original domain consists of documents. After the conceptualization, a knowledge base consisting of documents and elements (to the left) are the result. Another view is to regard the knowledge sources stemming from the elements only (to the right).**

Preserving original document structure along with an identification of elements, are useful as it provides flexibility and the choice to suddenly "turn off" the adaptive guidance provided from the system. The other view is that of the left side from the figure, indicating that only the knowledge sources stemming from element concepts should be regarded when generating adaptive documents. Note that in this case the document concepts are still necessary for the identification of the parent relations. We conclude that more empirical research is needed in order to evaluate the positive and negative consequences of the separation of document concepts and element concepts, along with the possibilities for adaptation.

### 6.3.4 Clean up the mess before the guests pay you a visit

According to Zibell, Klare's notion of "useful information" is still useful for web designers today, almost 40 years later [*Zibelloo*]. For traditional systems, knowledge of the user's knowledge level would help the author to suggest the correct depth of concepts and organization of content. The resulting domain will be static. For an adaptive hypertext system, the dynamic organization depends upon how well the DM can be constructed. Starting with the DSL construction, the following processes of analysis, element division and generation of relations are up to the system, with a resulting domain model as the gift to the author. As seen, the number of relations will probably get out of hand, but this does not mean that those found are not useful for the author. Obviously, the task of removing incorrect or unwanted relations and/or concepts, are a task a lot more pleasant for the author, than creating an entire domain model from scratch. Additionally, recall that all the important elements are sorted out of their original files and saved in a knowledge base, with the references kept intact in the `kw_source` facts. In other words, the present prototype of the system will indeed be expected to help the author in preparing a static domain for dynamic, adaptive presentations.

Due to the possible imperfect outcome of the proposed DM and its implicit assumption of well organised, consistently marked up documents, we would also have to require a proper use of tags in order for the AHS to perform at its best. At least the author should prepare existing documents for the system initially before the automation starts[1]. For future adaptation we propose that new documents should be composed with some guidelines in mind:

- Hyperlinks should be labeled so as to predict the content of the target.
- The domain should be well structured and decomposed into smaller documents.
- Knowledge sources on the same subject should be varied in different elements like lists, images, tables, text etc. in order to enrichen the knowledge base.
- Emphasizers should be used with caution.
- If applied, the title element and meta information should predict the content of each individual document well, and be varied among the domain documents.

The prototype does not handle poorly tagged documents, nested lists, and simply skips advanced features like scripts and css.

1. The Tidy program is available from www.w3.org and helps to fix incorrect use of mark up tags.