Implementation

In order to be able to assess the proposed conceptual model and framework for SWeMPs, a proof of concept system is implemented that demonstrates the core functionality of the knowledge-integrated multimedia generation process. This implemented system is developed with the Semantic Web in mind and will be used for the evaluation of the proposed approach.

While the framework of the Semantic Web-enabled Multimedia Presentation system has been developed using established design methodologies, it remains necessary to build a (skeleton) implementation of the system to act as a 'proof of concept'. This implementation is based on an established software development methodology, drawing upon the UML models of a system framework (Figure 4.2) and application process (Figure 4.5). In making concrete implementation decisions we take care to support integration at every level with the Semantic Web.

In this chapter the implementation that has been made within the context of the PhD research is explained. The implemented system is used for the evaluation of the following chapter. It is to be noted that a full implementation (including the extent of stability, scalability or efficiency that would be desired in a piece of commercial software) has not been possible within the time frame of this research. Rather, a minimal system framework has been produced which is sufficient to demonstrate the core functionality of SWeMPs. Where remaining aspects of the system have not been implemented, we draw upon the design decisions of the previous chapter to infer that requisite functionality would be possible. At this stage however, the intention is to show the fundamental plausibility of the proposed approach and its value to the Semantic Web and multimedia generation task, without attempting to address every possible issue. The implementation is available as open source software for further, possibly collaborative, ongoing development[42].

5.1 Software development methodology

For this implementation, the Rational Unified Process (first introduced in section 1.5) [Kruchten,2003] is the chosen software development methodology.

As an evaluation of the development activity itself, we mention the best practises as captured in RUP. These best practises are drawn from the experience of failed software projects and are considered to represent the best means to overcome the most common problems that can occur in the development cycle.

[42] As a project on the SemWebCentral website http://projects.semwebcentral.org/projects/swemps/

The implementation described in this chapter is shown to exhibit each of these best practises:

(1) *Develop software iteratively* – as has been previously noted (cf. appendices), this software development cycle represents a third iteration, and in each cycle alterations were made progressively during implementation
(2) *Manage requirements* – a set of detailed requirements, drawing upon the initial problem statement, inception of a possible approach and consideration of prior work (particularly the SRM-IMMPS), has been given in chapter 4.3.1.
(3) *Use a component based architecture* – component-based approaches produce a system that can be extended, promotes re-use and is intuitively understandable. The use of UML (which is object-oriented in nature) and particularly a component diagram for the SWeMPs architecture demonstrate a core commitment to a component-based approach from the design phase.
(4) *Visually model software* – the use of models aid greatly in the understanding of a system and hence support its implementation. UML has been used in the design of SWeMPs to graphically represent its proposed architecture and the multimedia generation process.
(5) *Verify software quality* – Quality assurance, while not as requisite in this case as would be in a commercial project, is still an aspect that will be taken into account in the ongoing development of SWeMPs. The evaluation of the skeleton implementation is a first step towards quality assurance. Bug tracking and so on as part of the public distribution will encourage continued quality in the development.
(6) *Control changes to software* – again, the initial implementation is seen as a proof of concept and change tracking has not been a major issue. However in the continued development of the system this will be included, as shown by the establishment of SWeMPs as a CVS-based project.

5.2 The rules-based system

As has been argued in the previous chapter (section 4.3.5), the application will be built as a forward-chaining rules-based system.

The most common logic programming paradigm for rules-based systems is Prolog (PROgramming in LOGic)[Shapiro,1986]. Prolog is typically used in artificial intelligence applications such as natural language interfaces, automated reasoning systems and expert systems. It uses a subset of first order logic that is restricted to allowing only Horn clauses, that is, clauses of the form

(*p* **and** *q* **and … and** *t***) implies** *u*

Notably, the subset of first order logic used by logic programs such as Prolog differs from the subset of first order logic used by Description Logics. Furthermore, Prolog programs operate according to the Closed World

Assumption (anything that can not be proven true is false) while Semantic Web approaches such as OWL take the Open World Assumption (anything that can not be proven true can not be said to be false). In terms of the realisation of the multimedia generation process, this can be seen to be an advantage as while it is realistic to expect that knowledge on the Web is always incomplete (given the dynamic, constantly changing, open environment of the Web), it is necessary within the multimedia generation process to be able at times to say definitely if something is to be considered true or false. Likewise, in the Open World it would not make sense to complete a sub-process by saying that there is no further knowledge on a particular concept as Open World would say that it can never be known if there is not some further knowledge to be found. Hence in SWeMPs the choice of a Prolog paradigm for the internal knowledge activity has both pragmatic and conceptual meaning.

Pragmatically, the Semantic Web is not yet mature enough in the rules layer to be able to be applied to expressing the rulebase of the SWeMPs application. Rather, Prolog with RDF/OWL extensions allows the use of the much more established Logic Programming (LP) paradigm with Semantic Web data. The current rules layer of the Semantic Web is under debate, particularly in terms of which forms of reasoning should be allowed and trying to maintain compatibility with the RDF and OWL layers. The choice can be characterized as being between Datalog (LP) approaches, e.g. allowing some forms of closed world reasoning, and RDF/OWL compatible (DL) approaches, which semantically are difficult to converge and could result in two separate development tracks [Horrocks,2005]. A Semantic Web Rules Language (SWRL) has been published as a W3C Member submission [Horrocks,2004] which combines OWL Lite/DL with Datalog. Hence the closed world approach, which we find useful in a semantic application such as SWeMPs, may yet be able to converge with a future RDF/OWL compatible Semantic Web rules layer. This activity continues to be promoted through the W3C Rule Interchange Format (RIF) Working Group [W3C,2006].

Conceptually, we can say that while the Open World Assumption is correct for the Semantic Web, the selected knowledge which is made available to the SWeMPs rulebase through the knowledge base must be then considered under a Closed World Assumption (while the world continues to change out there on the Web, in here in the SWeMPs process it is considered frozen at the current model of knowledge so that decisions can be made without ambiguity). While considering a particular set of knowledge under an Open or Closed World Assumption is a matter for the query engine used (and hence closed knowledge is modelled in the Prolog working memory and queried through Prolog unification, while open knowledge remains in the RDF knowledge base and is queried by a Semantic Web reasoner), it also makes a difference in the range of logical expressions that are possible, e.g. negation. Hence in SWeMPs care is also taken in terms of which knowledge from the (Open World) knowledge base is mapped into the (Closed World) working memory where it can be queried

under the Closed World Assumption. One effort to resolve this issue is DLP – Description Logic Programming [Grosof,2003] - which defines a common subset of logic shared by both Description Logics and Logic Programming and hence provides guidelines to mapping between both paradigms.

In Prolog, facts are n-ary predicates – a predicate *p* with an unlimited number *n* of atoms *a* applied to it, i.e. $p(a_{1,...,n})$. A rule is in the form

head :- *body*

where *head* is a single fact, *body* is one or more facts and the meaning of the rule is that if *body* is true (each fact unifies with a fact in the working memory) then the system can infer that the *head* is also true (i.e. add it as a fact in the working memory).

In SWeMPs, we consider the fact base maintained by the rules-based system as separate from the knowledge base (i.e. an instantiation of the conceptual model introduced in the previous chapter). The former is stored within the rules-based system and is expressed in terms of the rules-based systems logical formulism (Prolog) and uses concepts which have no explicit formal meaning outside of the system. The latter is external to the rules-based system and is expressed in terms of its own logical formulism, based on the Semantic Web, and drawing its concepts from the SWeMPs conceptual model. The inference engine is the rule engine that the application will be built upon, extended by the components identified in the framework such as a reasoner to allow for DL-based querying upon the knowledge base.

In our case, we use the unification of the fact base with the rules in the system as the basic means to trigger actions, in that some facts in the rule body to be evaluated are actions upon other components which are true if and only if they can be performed without failure. These actions are wrapped in special rules that execute API methods on the other, external, components. In other words, the call to other components is “disguised” to the rule engine by rules whose body is evaluated true if the methods executed on the component return true (do not fail) and result in a fact (the rule head) added to the working memory both as a signal that the component interaction took place and as a means of carrying the interaction results into the rules-based system (in the form of the atoms within the fact, given that the Prolog implementation does not restrict atom types). This is even though the components operate in a different paradigm (e.g. Semantic Web data models for the reasoner, Web Service descriptions for the service planner).

It is an explicit aim of the implementation to support the heuristic solution of multimedia presentation goals through declarative programming. That is to say, the implemented application must demonstrate an iterative solution to a given information request by examining all possible execution routes (with regard to knowledge, content and presentation), resolving conflicts (matching rules and passing control when rules fail or bottleneck) and concluding with a “best case” result. Prolog systems support this in that, given failure in a rule according to one

combination of variables, the Prolog interpreter will backtrack (to the last step before failure occurs) and try again to evaluate facts with another combination of variables (i.e. the interpreter tries all possibilities before giving up). Likewise, it is possible to declare rules with the same head and different body: if the evaluation of the head fact resolves to false on the basis of one of the rules, the interpreter will attempt again using another of the rules.

As we seek a rules-based system which will be able to be integrated with other components, a common programming language environment is required that is irrespective of the platforms upon which an application may be deployed. Hence, while within the system itself and system components logic-based programming languages form the underlying implementation model, Java is chosen to function as the programmatic "glue" between the otherwise distributed and heterogeneous components. Java is seen as a good choice as potential component choices are likely to offer Java-based APIs. A logic programming environment is required which can incorporate Java calls (to the other components), and Prova[43] was chosen. Prova is a rules-based scripting system using both Prolog and Java constructs to combine imperative and declarative programming styles within the Java runtime and with access to all Java packages.

5.3 Component implementation

The component-based framework of SWeMPs was illustrated by an UML component diagram in Figure 4.3. In this section we consider the concrete implementation of the components for the Semantic Web related aspects of the system architecture. The aspects of the service planning and multimedia modelling are then discussed in some more detail in the following sections.

5.3.1 Ontology creation

The conceptual model needs to be formalised as an ontology and other ontologies may need to be prepared for use with the SWeMPs system. While the model has already been specified in Description Logic (section 4.4.3), a tool is required to realise this specification as an ontology in a Semantic Web language such as OWL. Protégé[44] is not only one of the best known and well established ontology editors in the field but is extended by numerous plug ins to provide additional functionality needed or preferred by the SWeMPs framework, including OWL/SWRL support (for the Semantic Web interoperability) and (through plug-ins such as OntoViz) ontology visualisation.

[43] http://comas.soi.city.ac.uk/prova/
[44] http://protege.stanford.edu

5.3.2 Ontology population

In the SWeMPs framework, a knowledge base is foreseen as the external, explicit representation of the knowledge required for a given multimedia presentation generation task. The knowledge base consists of instantiations of concepts in the SWeMPs ontology. Populating ontologies is a problematic task, given the need to identify the instances required together with their properties and the possible scale of instances that may exist.

Protégé can be used for the manual creation of individuals based on the SWeMPs ontology, though as a tool it is not ideal for this task, especially as one would presume that the ontology population could be a task done by non-ontology experts (as opposed to the ontology creation above). There is work in (semi-)automated population strategies which could also be examined to ease developer effort, while we also note that SWeMPs is modelled in such a way to maximise the re-use of instances from other knowledge sources and to minimise the need to instantiate directly a large number of SWeMPs concepts (this will be shown more in chapter 6). Hence, for example, instances of services could be harvested from a Web service directory or better still, the look-up at the directory itself modelled as a single service instance in the knowledge base which is used to identify other services meeting the current system processing need.

5.3.3 Ontology storage

SWeMPs will import dynamically external ontologies during the multimedia generation process, so the size of the knowledge base being employed by SWeMPs can change significantly during its execution. In-memory storage of the ontologies being used in SWeMPs can become costly; hence generally some sort of persistent back-end is used to provide a storage solution to Semantic Web applications.

There is currently a lack of OWL specific storage systems. Rather, an alternative storage method could be used (database, database front-end for RDF, RDF specific triple store) as long as it can still be integrated with OWL specific reasoning capabilities. We look at the possibilities of reasoning tools in the next subsection.

Sesame[45] is a well known and established RDF(S) front-end to a database system (using MySQL). However its API and reasoning capabilities are limited to RDF(S)[46], and hence without a further development to include OWL level functionality can not be considered at this time for SWeMPs. Kowari[47] is an interesting implementation of a specific RDF triple store, with accompanying

[45] http://www.openrdf.org:80/
[46] There is an OWL DLP inferencer for Sesame which supports a few OWL constructs in a naïve fashion on top of RDF(S).
[47] http://www.kowari.org/

improvement in efficiency. The developers state their intention of extending the store to support OWL DL, but this work is not yet complete.

Jena[48] provides a Java-based framework for interacting with RDF and OWL knowledge stored in a RDBMS, and permits OWL level reasoning to be applied to that knowledge. The 100% Java approach of Jena makes it appealing for the realization of OWL level functionality in SWeMPs, especially as it offers an abstracted API for accessing stored knowledge and reasoning over it and leaves the actual relational database and reasoner implementation to individual developer decisions without forcing an alteration of the Jena code. Additionally, examples of Jena integration with the rules-based scripting of Prova are provided with the Prova download, showing that these two technologies can be combined in the SWeMPs framework.

At this stage of research, a simple Jena approach is taken, storing the knowledge base in memory. However it is clear that this implementation approach can be extended in future to use a dedicated storage solution and, if the Jena API is not available or functionally limited for use with that solution, that another API – preferably Java based – could also be used by SWeMPs.

5.3.4 Ontology reasoning

Interacting with knowledge based on an ontology requires access to a reasoner component which can proof the consistency of inserted statements and infer new facts from the statements present in the knowledge base. In SWeMPs, the reasoner will be working with a potentially dynamically changing knowledge base and will need to be stable enough to handle reloading of the knowledge base (rather than relying on a cached model). Despite the core importance of reasoning in the Semantic Web, current reasoners often demonstrate performance problems, particularly regarding scalability. While SWeMPs should aim to minimize problems by retaining knowledge in the active knowledge base only for the duration of its relevance, current reasoning support with promises of stability and scalability do exist.

Instance Store[49] is a DL reasoning solution for large numbers of individuals. It consists of a backing store (Oracle, MySQL or the 100% Java implemented HyperSonic) accessed through JDBC, a reasoner (Racer or FaCT) accessed through the DIG interface and an ontology (without instances). More recent versions of Instance Store include OWL support, but the major limitation is that it performs only “role-free” reasoning over individuals. Hence this work is interesting for overcoming future issues of scalability but insufficient for the initial realisation of the SWeMPs framework.

[48] http://jena.sourceforge.net/
[49] http://instancestore.sourceforge.net/

Jena, which supplies a Semantic Web API that is well defined[50], includes a dedicated reasoner, but other reasoners which provide more expressive and efficient reasoning capabilities can be alternatively plugged into the Jena framework. The Jena OWL reasoner is an extension of the existing RDFS reasoner, which results in an incomplete OWL based implementation. All OWL Lite constructs are covered, with the critical constructs NOT covered in the existing reasoner beyond OWL Lite being *complementOf*, *unionOf* and *oneOf*. The developers acknowledge stability and scalability problems, and advise caution in its use.

While the Jena reasoner is the obvious choice as an initial solution for reasoning support given the choice of the Jena API, it is insufficient for a stable system. Hence, while the proof of concept system developed for this thesis relies on the Jena reasoner, we plan to experiment with other reasoners in the ongoing development of SWeMPs.

5.3.5 Ontology query

Access to knowledge in a knowledge base is through queries. An appropriate query language is required that can express the form of queries that will be made by SWeMPs in deriving a multimedia generation process from the provided conceptual model.

In Jena, the query interface to the ontology uses as a query language RDQL [Seaborne,2004]. If another reasoner were to be used, the query language could also differ – in fact, it is expected that the SPARQL [Prud'hommeaux,2006] proposal will establish itself as the 'official' RDF query language, at least at the W3C. We note that for SWeMPs it is no problem to switch the query language, as this is abstracted by the query handling and reasoner components of the SWeMPs architecture. For example, ARQ[53] has been developed as a SPARQL processor for Jena.

RDQL is a SQL-like language which is executed within Jena through a query string, where the format of the query is SELECT ... WHERE ... (AND ...) (USING ...). As the form of the query is a single string, the following conventions are used to differentiate between types:

- URI References are enclosed within <…>
- Free variables are prefixed with ?
- Bound variables are enclosed within '…' (i.e. they are unified as strings even though they may also be integers or other datatypes)

[50] http://jena.sourceforge.net/javadoc/index.html
[53] http://jena.hpl.hp.com/ARQ/

We consider the task of passing the request to the SWeMPs rulebase as the task of the query handler component and the resolution of that request through executing the query – as RDQL or whatever is supported - upon the conceptual model as a task realised by a dedicated query rule in the rulebase using the API of the reasoner component. The dedicated query rule is discussed in 5.7.1.

The query handler forms the query by binding (or not binding) values, as well as their types, to an abstract statement – which, following RDF convention, can be seen as a triple of the form <subject, predicate, object>. The statement can be understood as the users' desired information, in that it seeks to state some fact about something. It is an information request because some aspect of the statement is missing, i.e. the user is missing some part of the desired information. The mapping from the expression by the user of which information they desire (which would be through some user interface which communicates to the query handler of SWeMPs) to this incomplete statement is specific to the query handler component used which is implemented to handle input of a certain format from the user interface application.

The purpose of the query handler is to determine the binding of values to the subject, predicate and object of the abstract statement. The bindings are also typed and could be individual values or a set of values[54]. Sets of values map in the LP paradigm to lists. Value types (which would be implementation specific from the requesting application) are mapped to Semantic Web concepts and passed to the rulebase as URIs[55] or datatypes according to the XML Schema specification[56]. Those values that do not receive a binding are left unbound, and hence are mapped in the query as variables (which, nevertheless, can be typed). In other words, the abstract statement is a means to provide a representation of the query that is not query language specific. The rulebase execution takes the abstract form of the query as a parameter when called from the query handler component. The dedicated query rule within the rulebase performs the further mapping of this abstract form to a concrete form based on which reasoner is being used: RDQL or some other query representation (e.g. SPARQL).

It is a fair comment that RDQL, SPARQL et al are 'RDF' query languages, however this does not make them invalid for querying an OWL ontology as it is the reasoner component, if it supports OWL constructs, that determines the inferable statements in the knowledge base. These statements, even if they also contain OWL, can be modelled in RDF (as OWL can be represented within the RDF data model) and hence matched by a RDF query language.

[54] How the query handler stores internally the set of values is implementation specific
[55] Typed as instances of java.net.URI within the rulebase.
[56] All the principal XML Schema datatypes have natural mappings to Java types. Date and time datatypes have Java types defined in the javax.xml.datatype package

5.4 Service planner

It has been argued that, for a dynamic process in which it is not possible to determine all functional needs in advance, a possible solution is to dynamically locate and execute external services, which encapsulate a given functionality, at run time when the specific functionality is required. This saves a system implementation from needing to explicitly code for every possible functional requirement in its lifetime. As an example, a multimedia system may need to convert between resource media types. As the possible set of media types grows, so the functionality encoded within the system must grow to cover each possible conversion possibility. Furthermore, certain conversion tasks may occur very rarely and it seems wasteful to have this conversion code explicitly maintained within the system. Additionally, the conversion may already have been implemented in an existing program and it would be better to simply re-use this code. The Web Services infrastructure is focused on providing such capabilities by enabling heterogeneous systems to communicate with one another regardless of their underlying implementations by defining a standardized interface for exchanging messages between them (e.g. calling a certain method on the other system, and receiving the response). In particular, Web Services focus on realising this infrastructure on the Web, so that any Web Service is reachable to any system which is connected to the Internet. To further support Web Service usage, a directory service can be invoked to find suitable Web Services, QoS considerations can be considered (i.e. availability, response time) and communication set up between services through the initial exchange of description files (which define the interface to the service, i.e. methods, their definitions and the required parameters to be passed and their syntax).

In the current Web Services infrastructure, the directory is encoded in the UDDI standard, service descriptions in WSDL and messages exchanged using the SOAP format [Curbera,2002]. However, examination of the Web Service directory, parsing of the Web Service description and the formulation of the SOAP messages is a manual task, in which a developer must write code specifically for the service to be called. In the light of the Semantic Web effort to define machine-processable data formats, efforts are being made to apply this to Web Services, so that their discovery, description and invocation might be expressed in a machine-processable form and hence automated in computer systems. Semantic Web Services, as the field has become known, is aiming to achieve precisely this [McIlraith,2001].

In the implementation, we use OWL-S [Martin,2004] as it is based upon the OWL language and has already a number of open source tools for editing, discovery and execution which can be used. However, the field is still open – another activity, WSMF [Bussler,2002], seems to be more comprehensive in its work and it seems likely that rather than result in two competing approaches the two groups will converge on a single agreed standard. At present, however, the latter work is still evolving so the stable OWL-S 1.1 will be used. Architecturally, it is

not a problem to support some future emerging Semantic Web Service standard as the specifics of the Semantic Web Service interaction is abstracted from the rulebase using the dedicated service rule and service planner component.

OWL-S acts as an upper ontology for creating service descriptions. It consists of three key classes, the ServiceProfile – which provides an advertisement of what the service does, the ServiceModel – which provides a description of how the service works, and the ServiceGrounding – which provides the means to interact with the service. These classes can be understood as realising the discovery, description and invocation of the service respectively.

The Service Profile provides a means for representing the provider of the service, the function of the service and the characteristics of the service. Functionality is modelled in two aspects: the information transformation (represented by inputs and outputs) and the state change produced by the execution of the service (represented by preconditions and effects). This is often expressed together as an 'IOPE'. Inputs and outputs have a parameterType parameter which identifies which (OWL) class their values belong to. Preconditions and effects are represented as logical expressions which also identify their representation language (e.g. SWRL, KIF).

The ServiceModel provides the details for interacting with a service, which is viewed as a process. An atomic process expects one message and returns one message. A complex process requires the exchange of a set of messages and maintains state as the interaction progresses. In SWeMPs, we expect to work primarily with atomic processes. The process is defined in terms of IOPE, typically in more detail than in the ServiceProfile.

Finally, the ServiceGrounding provides the means for actually exchanging messages with a concrete instance of the service. Groundings are based on a WSDL service description, which identifies the URL of the service, the syntactic form of the messages received and sent by the service, and the datatypes of the parameter values. Relations are defined between the OWL-S process and the WSDL operation, the OWL-S inputs and outputs and the WSDL message, and the OWL-S parameterTypes and the WSDL abstractTypes. In other words, a means to translate between the semantic and syntactic views of the service is provided.

We implement a three step process (discovery, selection, execution) within a dedicated service rule in the rule base, applying backtracking to ensure successful service interaction (e.g. after the first discovery step, if the second examination step fails because the selected service requires something that can not be provided, then the system backtracks to the first step and selects a different service and tries again). The service rule interacts with the service

planner that is chosen for the SWeMPs implementation, which could be (initially) using the OWL-S 1.1 API[57].

Considering the operation of the SWeMPs multimedia generation process (according to the UML activity diagram in Figure 4.5), there are three cases in which the use of external (Web) services is foreseen:

- Service “look-up for other metadata”
- Service-based “mapping between ontologies”
- Service-based “resource adaptation”

The SWeMPs conceptual model supports specifying properties on Services which indicate a namespace or media type handling service (i.e. an ontology mapping or resource conversion service). This is a form of shorthand for regularly requested services to save on the reasoning overhead of querying the entire service description (directory). A classification of namespaces and media types may be useful for reasoning over these services, e.g. that a namespace is *partOf* another namespace or that a media type is a *specialisation* of another media type. Services which extract knowledge about specific domains could be modelled by having the namespace as value of the handles-namespace property, and services which support adaptation of specific media types could be modelled by having this media type as value of the handles-media-type property. However, other cases require the specification of different namespaces or media types for the input and output of the service, i.e. mapping between two ontologies requires the namespaces of the input and output ontologies; converting resources requires the media type that is input to the service as well as the type which is output. Hence, we can extend the SWeMPs conceptual model to differentiate input and output properties and define a simple methodology for service discovery through the values of those properties of a Service in the conceptual model:

Service type:	from-media-type	to-media-type	from-namespace	to-namespace
Resource adaptation	X	X		
Resource conversion	X	Y		
Knowledge retrieval			X	X
Knowledge mapping			X	Y

The service metadata can be examined for the selected services for two reasons:

1. To determine if a selected service can be executed at all, e.g. to test that the preconditions of the service are met;

[57] http://owl-s-api.projects.semwebcentral.org/

2. To determine which service from a number of possibilities is preferably executed, e.g. to check the current QoS given for each service.

We illustrate this in Figure 5.1 in terms of a service execution call (the execute rule is detailed in section 5.7.2) and the mapping to IOPE in an OWL-S service model, using as an example services for resource conversion and adaptation, based on an example given in [Jannach,2005]. The KoMMa OWL-S framework introduced by [Jannach,2005] is a good example for implementing a SWeMPs Web service framework except that it depends on MPEG-7/21 XML syntax references for its input and output, rather than the SWeMPs approach of using semantic concepts specified by URI and defined in the conceptual model.

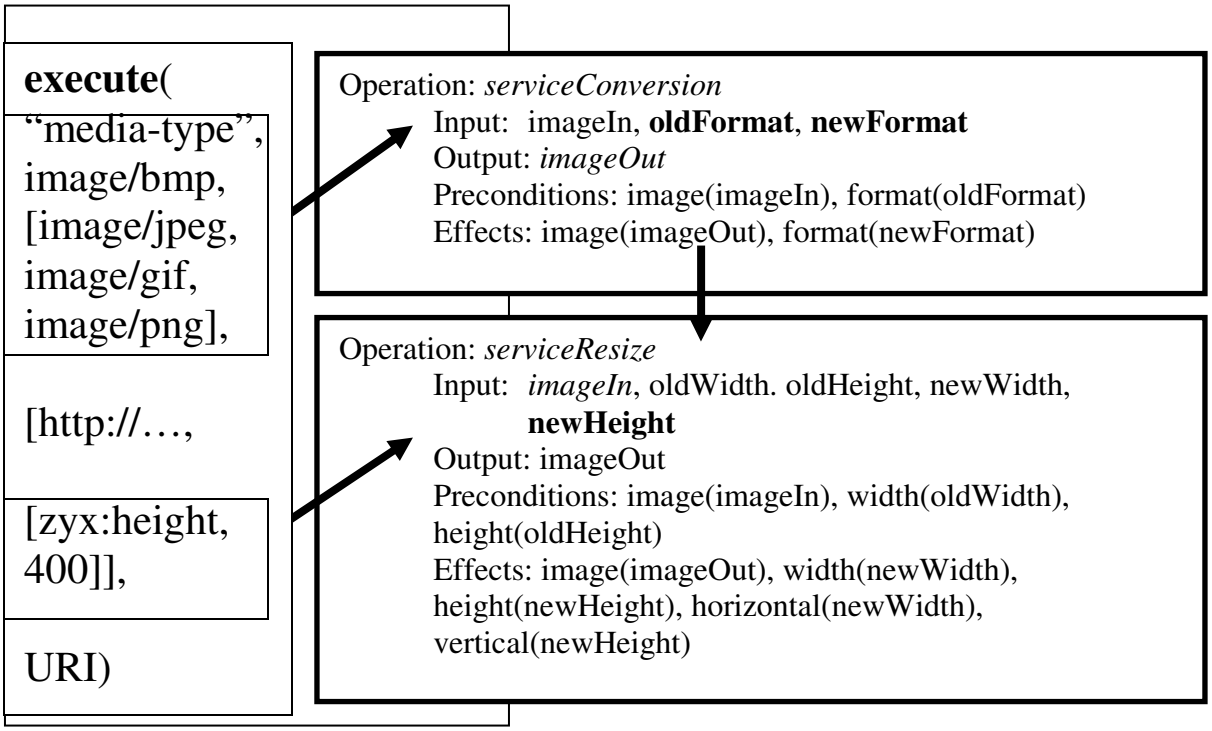

Figure 5.1 Sample integration with Semantic Web Services

The mappings here depend on the service parameters being typed semantically, i.e. the parameterType is an URI identifying a concept in the conceptual model which is mapped to the intended meaning of the parameters in the execution call. For example the MIME types given are mappable to the oldFormat and newFormat parameters (typed as swemps:MediaType) in the first service. As a height constraint is also given, a further service has to be called. By semantic type equivalence the imageOut output (typed as zyx:Image) of the first service can be used as the imageIn input of the second service. Note that in the second service extra parameters are given in the input (oldWidth, oldHeight and newWidth) that are not in the execution call parameters, so the service rule

constructs additional queries to extract the necessary additional knowledge (what is the width of the original image).

In the final step, the Grounding for the service is retrieved through the service metadata and used by the service planner to construct the message to the selected service, as well as to correctly parse the service response and pass it back to the rule base[59].

5.5 Multimedia modeller and formatter

An important issue for SWeMPs is the multimedia model and the multimedia modeller component. Fully abstracted from, and yet intractably linked with, the rules system the modeller is tasked with supplying the requisite functionality for operations between the rulebase and the multimedia model. Like the reasoner which connects the rulebase and the conceptual model, that carries with it a set of logical processing axioms (based on the knowledge representation it is intended to work with, e.g. RDF/OWL), the multimedia modeller must contain the set of minimum generic logical processing axioms for building a consistent and correct multimedia model. These axioms need to be specified just as logical axioms for knowledge representations must be specified. Additionally, the modeller needs to be able to deal with two further cases:

(1) The internal generic axioms for a model (as rules) are applied not only to the facts being generated from the rulebase but also to the execution-specific presentation constraints. These act as an additional input to the modeller and must be taken into account when carrying out consistency checking.

(2) The facts inferred by the rulebase for the model are insufficient for a multimedia presentation, as they are limited to stating which resources are relevant and not how resources are to be presented. This is a domain-specific issue and hence must be specified for each execution. Like the conceptual model, we suppose individual presentation rules can exist for different multimedia generation cases. These rules allow for inferring from the existence of resources (and their domain-specific properties and relations, tested against the conceptual model) to constructs which we have called “communicative abstractions” and which represent a set of constraints that apply between those resources.

In summary, the multimedia modeller acts like a reasoner upon a multimedia model, which follows a different set of axioms than a knowledge representation

[59] Selection and execution is handled by the service planner implementation, e.g. the OWL-S API, which re-uses the Jena toolkit to reason on ServiceParameters and uses the Axis WS package to execute services mapped to/from WSDL.

model. The multimedia model will contain a set of base constructs like objects, space, time and interactivity just like an ontology contains classes, properties, instances and literal values. The axioms will define the basic rules for a consistent presentation (e.g. that objects must all fit within the space available). The modeller accepts resource insertions as input, and builds up in the multimedia model a set of facts (determined by resources, their properties and relations to other resources). The set of facts is constrained by the consistency axioms including the presentation constraints specified in each execution. The model incorporates domain-specific constraints inferred from the facts about resources and their conceptual representation expressed in the knowledge base. When all resources have been inserted and all resulting inferences handled, a consistent set of facts which respects all the specified constraints can be formatted to a concrete multimedia presentation.

For an implementation, we needed to take into account four aspects of the multimedia modeller:

(1) The application level, e.g. implement as another rules-based environment like Prolog/Prova, a Java based application with Java based communication, an agent-based system using messaging or a structured modelling environment such as building an ontology with rules?
(2) The abstract model, i.e. the means in which the transitory and dynamically built model will be represented within the system. Possibilities include a working memory of facts, Java objects, an ontology or a tree-based structure like XML. These are themselves based on some abstract conceptualisation of a multimedia presentation which must be specified, preferably formally.
(3) A constraints handling component, incorporated as an extension to the base application. This uses general axioms and presentation constraints to check the consistency of the model being built, and solves the step from the looser abstract model to the more definite pre-formatted model. For the constraint handling, one could take a constraint logical programming (CLP) approach such as ECLiPSe or a constraint handler with a Java API such as Cassowary.
(4) A serialisation component which represents the internally stored definite pre-formatted model in some structured form which can be formatted by the formatting component into a final format multimedia presentation. With the expectation that the formatter could use XSLT to make the transformation, the serialisation could be XML based.

At this stage of implementation, we take a simplest-case approach while acknowledging that for a multimedia generation system this is a key aspect of the work which needs well developed components to handle the inherit complexities of multimedia presentation layout, and hence a key area for further development in SWeMPs.

Normally, the application level is realized in a tool called a “layout manager”. Examples of layout managers are GRIDS [Feiner,1988] and LayLab [Graf,1996]. Weitzman presents a tool which realizes layouts on the basis of a description using relational grammars [Weitzman,1994]. Rather than attempt at this stage an integration of this approach with an existing layout manager, SWeMPs will use a simple OO-based approach that interfaces with the rules-based system and manages the multimedia model in internal memory while interacting with a constraint solver to ensure consistency and derive final layout properties. However, it is accepted that an eventual integration with a dedicated layout manager would provide a more complete solution.

The abstract model must be able to separate the multimedia document from its final realization and representation. A number of multimedia document models are proposed in the literature, e.g. MATN and MADEUS (see 2.1.2 and 3.2). We have based the abstract model on ZyX (see 2.1.2 and [Boll,1999b]), as it defines a SMIL-like tree-based structure, which makes it suitable both for formatting into SMIL as final format and for representing within a program using a structured data model such as XML. Unlike SMIL, it permits media items to be placed in the tree without an initial binding to presentation properties (such as spatial and temporal positioning) and to group sub-trees into complex media items (supporting reference and re-use). It also supports metadata-based selection between alternatives in the tree to determine the final form of a presentation.

We use ZyX to define the abstract model’s syntax and semantics, with the model’s realization at the application level based on Java methods on a ZYXModel (Java) class. For the full formal framework of ZyX see [Boll,1999b]. For the implementation, this framework is modelled in OWL using Protégé as the ontology editor. Additional properties were associated with the ZyxModel (ontology) class to allow for specification of presentation constraints. The ZyX ontology and Multimedia Modeller Java classes are packaged with the SWeMPs source code[60].

In SWeMPs, the abstract model is combined with a constraints handler. The constraints handler will resolve the set of constraints expressed in the abstract model to a set of satisfiable concrete values which represent the final abstract form of the multimedia model prior to formatting (e.g. the actual spatial and temporal positioning of media items). The constraints can be expressed in terms of the abstract model (ZyX) syntax and mathematical operators. To illustrate how constraints are expressed, consider that two images are constrained such as one image *I* is to appear at least 10 pixels to the left of image *J* (see Figure 5.2). Given the properties *x* to represent the image’s location (measured from the top left corner) on the horizontal (x-)axis and *w* to represent it’s width in pixels, we have the constraint:

$$J.x - I.x > I.w + 10$$

[60] http://swemps.projects.semwebcentral.org

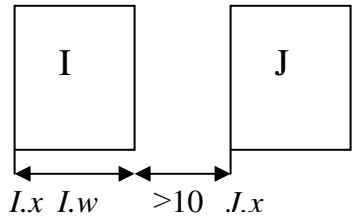

Figure 5.2 A sample constraint

As the application level is implemented within an OO-based paradigm, we chose Cassowary [Badros,2001], which is a Java based implementation, to realize the constraint handling. In Cassowary, the constraint example given above would be expressed as a ClLinearInequality object such as:

ClLinearInequality(CL.Minus(J.x,I.x),CL.GEQ,CL.Plus(I.w,10))

We build the ZyX model within the multimedia modeller component (calling constructors of ZyX elements given by the ZYXModel class and binding values to some of their properties) and refer to that model in the specified Cassowary constraints so that the constraint solutions are directly applicable to the bindings in the multimedia model. Default constraints can be given for many spatial, temporal and interactivity constraints and more complex constraints expressed in terms of a set of these default constraints. Thus it is possible to allow application developers to specify domain-specific constraints at a high level without requiring the generation of Java code.

The communicative abstractions are identified by URI, allowing for a decentralised system of definitions in which the use of unique namespaces avoids unintentional naming conflicts. Furthermore, the multimedia modeller should be extendible, i.e. that sets of communicative abstractions can be loaded into it. In our implementation we use Java, so abstractions are expressed as Java methods in a Constraints class which specify certain constraints upon two resources passed to them as parameters, and a simple text file can be used as an index read by the multimedia modeller at initialisation to map communicative abstraction URIs to the Java methods available to the modeller implementation.

Upon the conclusion of the model generation, the resulting multimedia model is serialized and formatted to the final presentation syntax by the formatter component. The formatting could be carried out in any means implemented within the formatter though for reasons of interchange and modification we prefer declarative to procedural approaches. Hence one possibility is to generate a XML file from the multimedia model and perform a XSLT based transformation. we consider SVG (in its formats for mobile devices [W3C,2003]) and SMIL [W3C,2001] as the main target formats for evaluation of the implementation. They are widely accepted (as W3C standard) XML based formats for the

representation of 2D multimedia presentations with a number of playback tools available.

5.6 Rulebase

The multimedia generation process has been illustrated in Figure 4.5 as an UML activity diagram and described in natural language. For the implementation, we expand this into a set of rules. These rules would be written in Prova, which takes a Prolog syntax with support for referencing Java classes and methods.

However, for illustrating the development of the rulebase in this chapter, we use a more accessible graphical notation based on ECA (Event – Condition – Action) rules. The ECA graphical notation [Berndtsson,2001] has been developed to aid software engineers capture the fundamentals of these rules in an application. It is based on modelling ECA rule features in UML statechart diagrams. A box containing the event is connected by an arrow pointing to a box containing the action, and the connection is labelled by the condition for the rules execution. In the diagrams, the text is written in a form of pseudo-code. FOR statements represent loops where the subsequent statements are evaluated against each value matching the condition of the FOR statement. The action takes place only if the entire condition is met.

Terms used are taken either from the conceptual model, are internal to the rulebase (such as Input and Constraints) or refer to components (ConcModel represents the SWeMPs conceptual model, hence “exists in ConcModel” means a query handled by the reasoner component and MultModel represents the abstract multimedia model, hence “inserted in MultModel” means a command sent to the multimedia modeller component).

Rule 1.

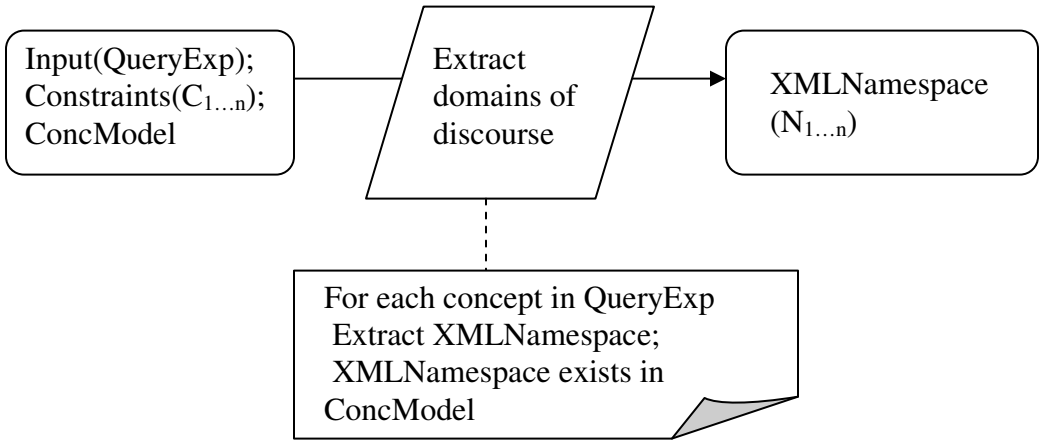

Figure 5.3 Rule 1 – extracting domains of discourse

The event triggering an initial rule to begin the multimedia generation process is the assertion in the working memory of the rulebase of an input, containing a query expression provided by the query handler component, a set of presentation constraints which are related to the resolution of that input (i.e. details of the user which caused the input and the device which generated it) and a reference to an instance of the SWeMPs conceptual model (i.e. the knowledge base) which defines the specific multimedia generation process to be performed for this execution. The query expression contains URI-typed constants or variables which represent the subject, predicate and object of RDF triples as well as optionally URIs to represent their types. The variables represent the values to be found to resolve the query. The presentation constraints could be expressed in any suitable constraints model but for implementation we suppose it will be a set of RDF triples, each of which expressing an individual constraint. A suitable format may be CC/PP [W3C,2004b]. A set of core extension properties used with SWeMPs for the presentation constraints, labelled CC/PPx, are presented in [Nixon,2005].

Given an abstract query which is expressed as a subject, predicate and object, SWeMPs extracts the part of each concrete URI in the query which identifies the ontology the term exists in rather than the part which identifies the term itself. Where a XMLNamespace concept exists in the conceptual model whose address is equal to the extracted URI, SWeMPs asserts that XMLNamespace as a fact in the working memory.

[62] http://www.dfki.uni-kl.de/frodo/RDFSViz/

Rule 2

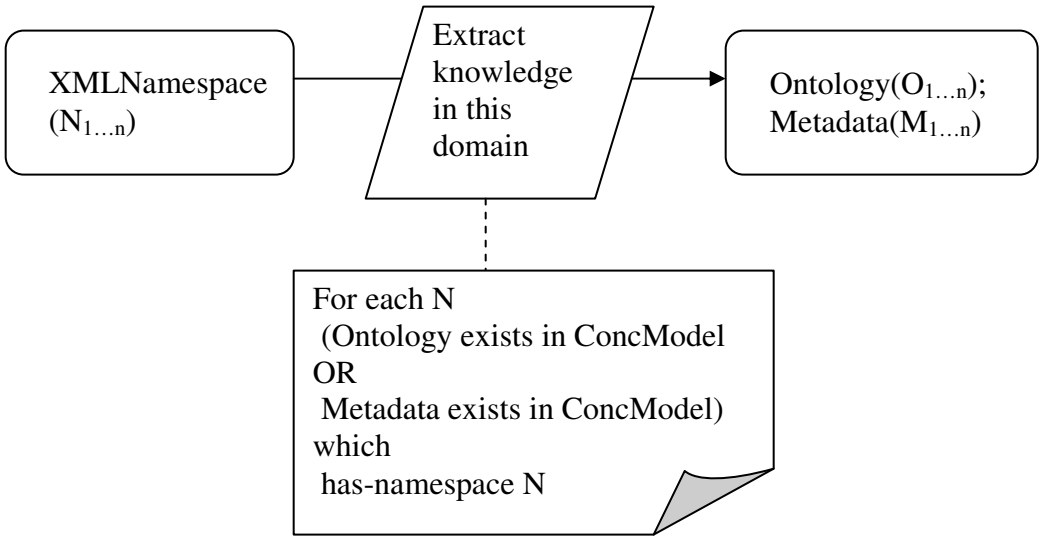

Figure 5.4 Rule 2 – extracting ontologies and metadata

Given that a XML namespace is asserted as a fact in the working memory, query the conceptual model for instances of Ontology and Metadata which "have this namespace". The relation has-namespace in the conceptual model is intended to signify that the semantic data model involves (to a significant degree) terms drawn from the given namespace. That this relation is explicitly given in the model allows the developer to determine which sources of knowledge are chosen for a certain namespace (i.e. for certain terms which may occur in the query expression). Of course, such relations might also be automatically determined by parsing the documents themselves.

Where such Ontology and Metadata matches exist they are asserted as facts in the working memory.

Rule 3

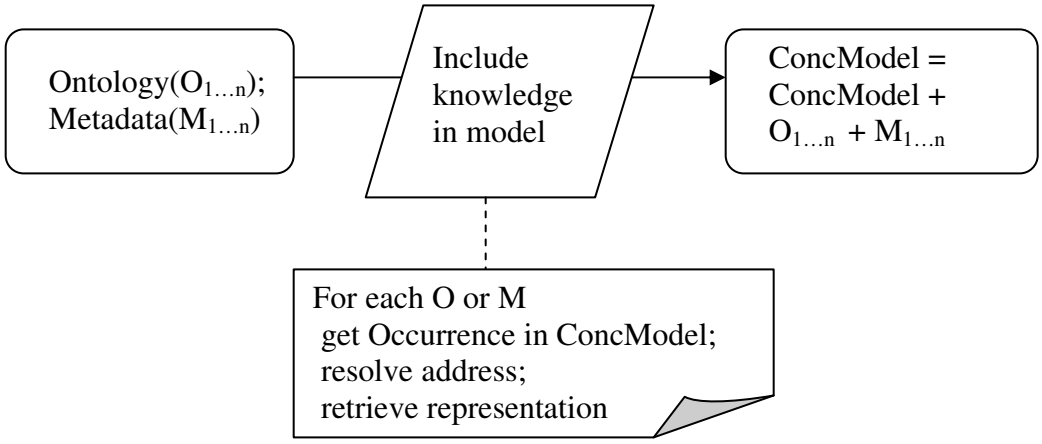

Figure 5.5 Rule 3 – Adding knowledge to the conceptual model

Given that an ontology or some metadata is asserted as a fact in the working memory, a representation of that ontology or metadata should be retrieved by selecting an Occurrence of the given concept in the conceptual model and resolving the address associated to it. The representation (assuming it is consistent with the representation of the conceptual model, otherwise assume a transformation of the representation also takes place e.g. through a Service) is integrated into the conceptual model and hence is made available for reasoning upon.

Rule 4(a,b)

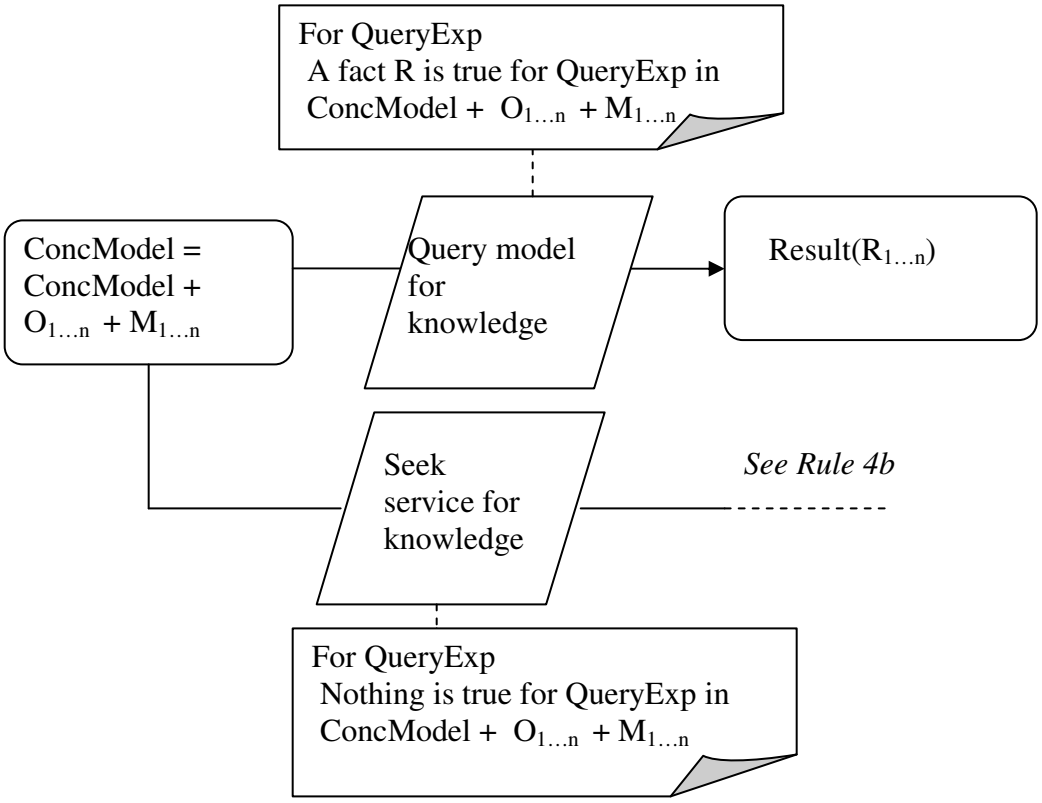

Figure 5.6 Rule 4a – get query results

Once the conceptual model has been completed with the insertion of representations of all the asserted Ontology and Metadata facts in the working memory, QueryExp (the query expression introduced as an initial input to the system) is evaluated against the model using the reasoner component. As the rule diagram above illustrates, two mutually exclusive outcomes are possible depending on which condition holds. In the first case, some fact R is found to be true for the given QueryExp in the actual conceptual model. The set of facts $R_{1...n}$ is asserted in the working memory. However, in the second case, no facts are found to be true, i.e. the reasoner returns the value null to the system when it makes the query on the conceptual model. In this case, a further condition is evaluated (see Rule 4b below)

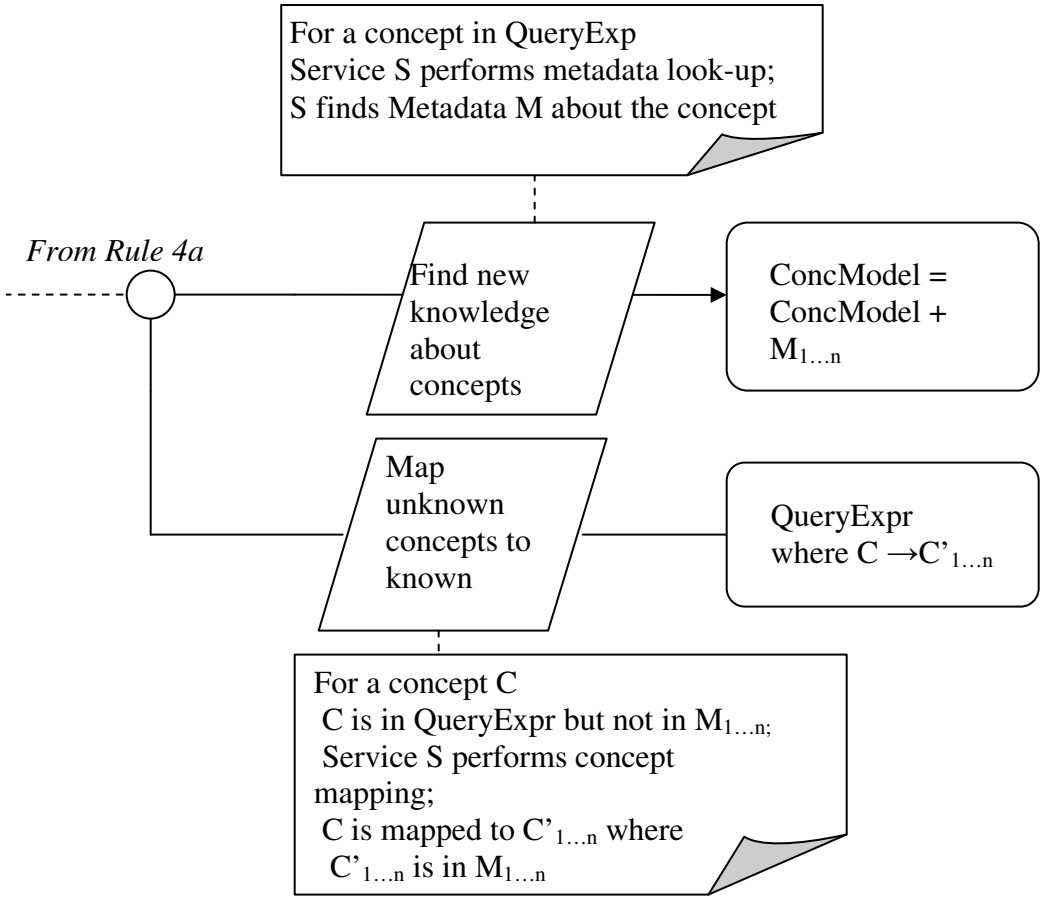

Figure 5.7 Rule 4b – using services to resolve queries dynamically

In this case, since it has been determined that the initial provision of knowledge through the conceptual model has not been sufficient (i.e. the set of metadata available to provide information about the concepts in the query expression) new knowledge and an understanding of that knowledge is sought. In this case, the rules are not mutually exclusive but concurrent. Given that most LP implementations evaluate rules not concurrently but sequentially allow the rule to check both conditions (in LP, we simply ensure the head of the rule is the same). While it is clear the upper condition should be evaluated first (as it determines the value of M, which is evaluated in the lower condition) it is not necessary to provide the upper condition with a higher priority (which, again, may not be possible in the logic program) – if the lower condition is evaluated first and fails, the system backtracks to the upper condition.

Given the existence of a service which can perform metadata look-up, the system executes this service for the concept in the query expression and includes the found metadata into the conceptual model (as new Metadata instances which are then included through the triggering of Rule 3). The new conceptual model is then once again queried (Rule 4a). If the query result set remains null and no more metadata is available through the service for inclusion, the remaining

condition is to attempt a mapping from 'unknown' concepts to 'known'. What is meant here is that the asserted set of ontologies $O_{1...n}$ represents the description of the domains from which the concepts in the conceptual model metadata are taken. In other words, for the system to be able to determine that a certain fact in some metadata matches the query expression, that query must be expressible within that set of ontologies, as the extent of 'known' domains which can be reasoned about within the system. Note this may involve both the concepts and properties related to a known concept within some metadata, as well as the concept itself (given the service which looks up metadata has been able to reason about the equivalence of concepts itself). Hence given a service which can map between ontologies, we execute a mapping from unknown to known concepts. This mapping takes the form of C → C' where C is the concept which is not contained in the metadata in the conceptual model. Given that this metadata draws from a set of ontologies $O_{1...n}$, a mapping attempts to find equivalences of C within this set and returns these as $C'_{1...n}$. In the query expression instances of C are equated by instances of C' (e.g. through the assertion of OWL *sameAs* statements in the conceptual model as a simple means, or through a semantic matching service for less trivial relations) and the query is once again performed. One can understand Rule 4 as an iterative process, where knowledge is acquired and mapped in terms of the 'known' domains until some result set $R_{1...n}$ can be acquired. If no result set can be found, the process terminates.

Rule 5

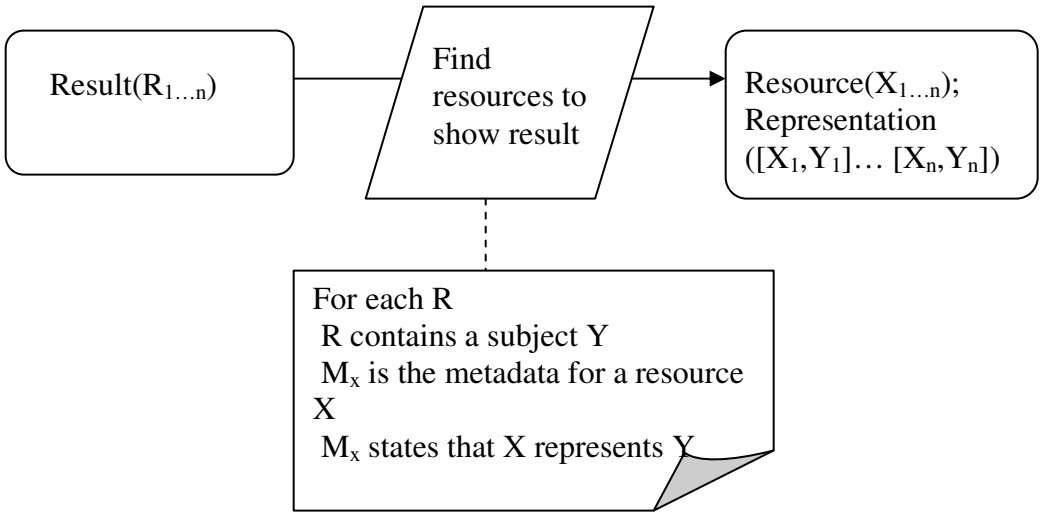

Figure 5.8 Rule 5 – finding resources for valid subjects

Given that a set of answers to the query expression have been found (note that we consider $R_{1...n}$ to be a set of complementary answers, i.e. that SWeMPs should try to communicate to the user as much as this set of answers as is possible as response to the query), a set of resources must be found that can be used to represent those answers. Each R, as a RDF Statement, contains three concepts which are understood to be the subject, predicate and object of the statement.

Among the metadata $M_{1...n}$ inserted into the conceptual model there is a set of resource metadata which is mapped into the SWeMPs model by ontology-specific mappings (through Rule 4b) that generate Resource instances with properties of representing a SWeMPs Subject Y. Where the subject Y is equivalent to a concept in R, and the resource metadata M_x describes a resource X that swemps:represents Y, then X is asserted as a relevant resource in the working memory. Note that X can still be related to its metadata through the conceptual model. The representation of Y by X is also asserted in the working memory of the system.

Rule 6

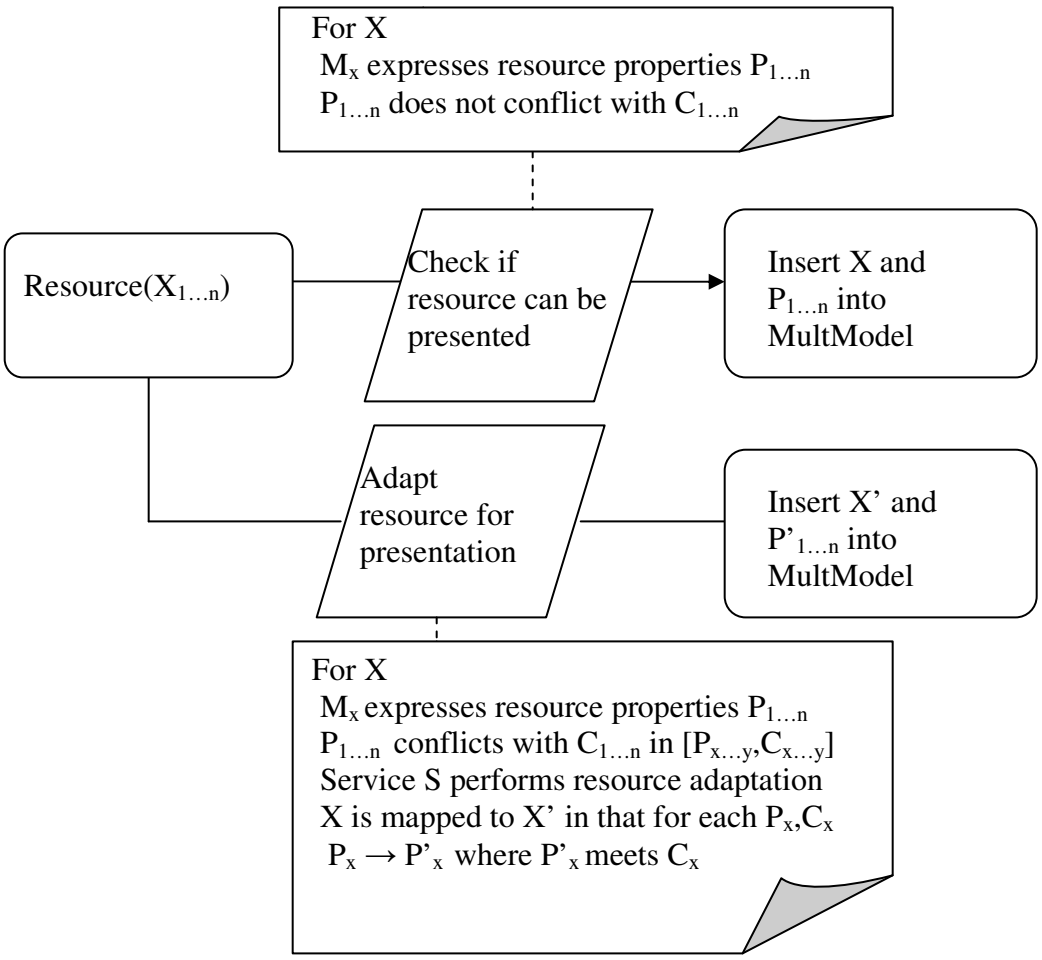

Figure 5.9 Rule 6 – adapting and inserting resources

The purpose of this rule is that for each resource asserted in the working memory, it is checked if the presentation constraints specified in the working memory are met by that resource in terms of the available metadata. If it can be determined that the resource does not meet the presentation constraints then a service is executed with the functionality to adapt resources, using as input a reference to the resource and the set of presentation constraints which conflict with the resource properties as expressed in the metadata. The output of the service will be a reference to an adapted copy of the resource.

To illustrate the matter of determining conflict between the presentation constraints and the resource properties (which can be understood equally as constraints upon its presentation), we assume that both are expressed as property-value pairs and take a set theoretical approach. Hence a conflict is seen as set non-subsumption – take as an example the case of display formats. In the presentation constraints we have:

ccppx:formatsSupported {GIF,PNG,JPEG}

and in the resource metadata we find that the resource described

dc:format BMP

Assuming the equivalence of properties (which can be expressed in the conceptual model to allow automatic reasoning over both properties as equivalent) and using set notation we have (Figure 5.10):

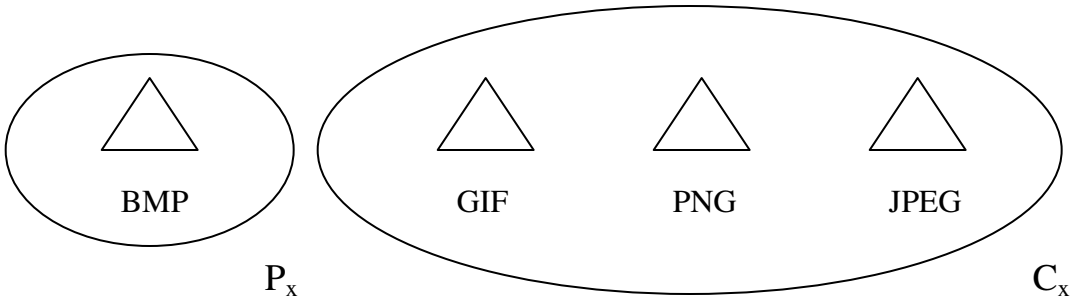

Figure 5.10 Set non-membership as test for adaptation

Hence this is seen as a conflict as BMP does not belong to the set C_x and conflict resolution is made by transforming the set non-member BMP to a set member one-of{GIF, PNG, JPEG}. In terms of the adaptation, this would mean executing a service which can take as input the resource in format BMP and return a resource in one of the other formats. This has been illustrated in the discussion on services in Section 5.4. The final resource and its properties (constraints upon its presentation) are inserted into the abstract multimedia model.

Rule 7

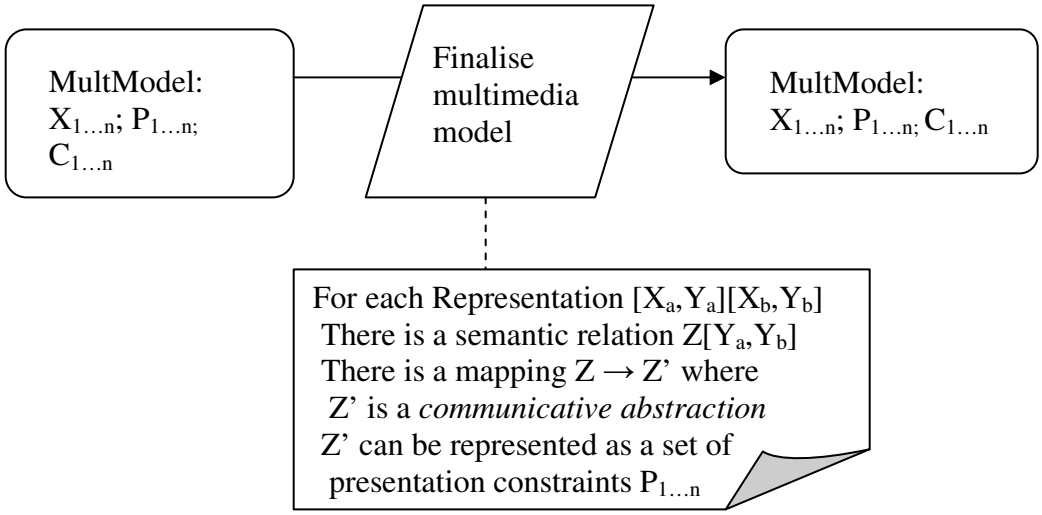

Figure 5.11 Rule 7 – finalising the abstract multimedia model

The previous rule is iterative in that it does not exit until all resources have been handled. The result is that the multimedia model contains a set of resources which each have some associated presentation constraints. While the resources themselves are assumed to individually conform to the input presentation constraints $C_{1...n}$, these constraints are also included in the model as some apply generally to the multimedia document as a whole and not just individual resources (e.g. screen size of the user's display device) upon the initialisation of the abstract model within the multimedia modeller. The model is still not yet complete for determining the final presentation as the constraints that should exist *between* the resources (as opposed to upon them) must also be determined.

The applicable condition here is that for two resources X_a and X_b there is also some semantic relation between the concepts they represent and a mapping from this relation (or, allowing ontological reasoning, an equivalent relation) to a *communicative abstraction*. The semantic relation can be determined by inspecting the conceptual model. The mapping to communicative abstractions must also be available to the rulebase from the conceptual model, and is introduced through Ontology instances which have relevant namespaces and contain SWRL [Horrocks,2004] rules encoded in the RDF/OWL syntax. Communicative abstractions should be understandable by the multimedia modeller, and hence are best based upon the predefined set of abstractions (otherwise the developer will need to extend the Constraints Java class of the multimedia modeller to support the newly introduced communicative abstractions).

Rule 8

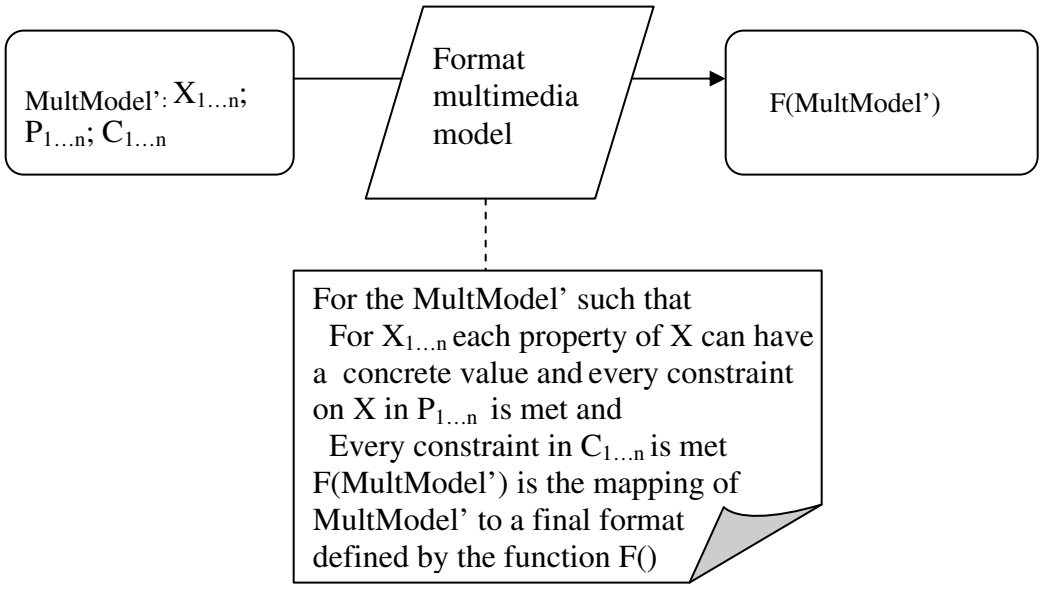

Figure 5.12 Rule 8 – formatting the multimedia model

The final abstract multimedia model consists of a set of resources, a set of (unary and binary) presentation constraints upon those resources and the input presentation constraints upon the multimedia document as a whole. This model needs to be solved in terms of finding a single model in which all constraints are met when abstract values (e.g. of resource spatial and temporal positioning) are mapped into concrete values. This solution model (assuming one exists, if not backtracking ensures that the operation of rules is reversed and a new set of possibilities is tried which can lead to a solution model) is mapped into a final format model through a function F() – which is implemented in the formatter component of the SWeMPs architecture. Naturally, the formatter may be able to provide a number of formatting functions, which then each describe their result in terms of constraints, and the function is selected which preserves all constraints upon the solution model. For example, functions may be available to format the model to SMIL 1.0 or SMIL 2.0 and if the solution model has the constraint

ccppx:final-format SMIL 1.0

then it follows that the SMIL 1.0 function is selected.

5.7 Component APIs

Having determined the components that will be used in the implementation and the rules that realise the process in the SWeMPs framework, we add definitions of how the rulebase shall interact with those components via a dedicated rule as an abstraction of the actual component API, which is accessed within the dedicated rule. This is done so that the rulebase (as an implementation of the process given in 4.3.4) is not dependant upon components in case their implementation changes. In other words, if a component in the framework is changed the developer only needs to alter the internal workings of the related dedicated rule to use the API of the new component. As long as the dedicated rule continues to respect the API defined here for the rulebase, the system will continue to operate.

The multimedia generation process foresees five dedicated rules which abstract component functions:

Conceptual model	`query(), include()`
Service space	`execute()`
Multimedia model	`insert(), output()`

It is important to note that the API given here must not be confused with those of object oriented programming languages. In the LP paradigm, no values are 'returned' as these are not methods that belong to any object. Rather as rules they are evaluated to true which means a set of valid values are found for all variables in the rule body. In that Prolog permits one to place any variable in the body also in the head of the rule, there is a means to pass values back from the rule into the rulebase. As a simple example, a rule to find the cube of any integer could be: `Cube(X,Y) :- X * X = Y.`

If somewhere else in the rule base, the fact `Cube(2,Y)` is given in a rule body, then in the evaluation of this rule the variable Y will be bound to the value 4, i.e. `Cube(2,4)` is evaluated to true and no other possible value of Y will evaluate the rule to true.

5.7.1 Conceptual model API

`query` abstracts the lower level implementation of the query interface to a conceptual model, and `include` abstracts the lower level method for importing knowledge into the conceptual model.

`query(subject, subjectType, predicate, object, objectType)`
`subject` may be of type URI, URISet or variable

`subjectType` may be of type URI, URISet or undefined `predicate` may be of type URI, URISet or variable
`object` may be of type URI, URISet, Literal, LiteralSet or variable
`objectType` may be of type URI, URISet or undefined

The purpose of the query rule is to pass from the rulebase to the chosen reasoning component a query on the conceptual model and to return a set of answers to the query. The parameters of the rule reflect the standard triple format of knowledge in the Semantic Web (RDF/OWL) – subject, predicate and object - while permitting sets of values to enable 'looser' query specifications. Furthermore as concepts in the Semantic Web are typed, both subject and objects can be further restricted when passed as unbound variables (i.e. where matching concepts are to be found) by being typed. This restricts query matches to concepts of that type or a subtype. The rule translates these parameters into a query string and executes a query with the available reasoning component. It then organises the results as a set of triples and returns this set to the rule base.

URIs represent concepts in the conceptual model (i.e. a RDF resource). In the rulebase these are represented by instances of the java.net.URI class. Literals represent a datatype value according to the XML Schema specification. These datatypes exist as first class Java objects in the rulebase.

URISet and LiteralSet are a list of URIs or Literals respectively encapsulated into a single parameter. In Prova, this is represented through lists (as in Prolog). A list contains either URIs or Literals, and not a mix of both.

Variable represents a computational variable whose value will be set as a result of the query. That value could be an URI, URISet, and if the variable was the object parameter, Literal or LiteralSet. Variables could be typed, which restricts the values bound to them to this type (for properties, subproperties will be automatically matched). In the case of untyped variables or when the subject or object are bound values, the type parameter defaults to '_' which represents a variable that does not participate in Prolog unification. When the subject of the query rule is a variable, a subject type can be given. Likewise, when the object is a variable, an object type can be given.

In other words, for a fact `query(X, Xt, Y, Z, Zt)` it shall be evaluate true where X and Z respect their types Xt and Zt and as a RDF triple can be matched in the queried knowledge base. So if there is `query(x,_,y,Z,zt)` – x , y and zt are bound values - then the variable Z will be bound to all (type-respecting according to zt) values in the knowledge base who exist as the object in a statement with subject x and predicate y.

The body of the query rule is dependant on the chosen Semantic Web API and reasoner, which in turn determine the query language used and the means of executing the query. In the case of RDQL, query strings always SELECT three

variables named k_sub, k_prop and k_obj. The reason for this is that the query result handling code is written generically, and does not know in advance which values are bound or unbound, and hence must be able to specifically reference all three variables. A sample RDQL query is given below:

```
"SELECT ?k_sub,?k_prop,?k_obj
WHERE ?k_sub <http://...> 'Lyndon' "
```

where the predicate is typed as an URI, the object is typed as a literal (string) and the subject is left unbound, i.e. should be bound to the response to the query. Where types are specified these are added as additional queries (subject rdf:type subjectType and/or object rdf:type objectType) which are joined with the first query.

`include(address)`
`address` is an URL pointing to a metadata or ontology file

The purpose of the include rule is to assert new facts, not in the rulebase (to do this, Prolog has a built-in predicate *assert*) but in the conceptual model. It does this by making a reference to a body of facts to be included (either an ontology or metadata instance) and exits successfully in that these facts are added to the conceptual model and are henceforth available to the reasoner.

URL shall refer to the location of a collection of facts made in some knowledge representation format. To support interoperability, we would expect this format to be RDF/OWL (otherwise mediation will need to be included in the process to map from other formats). The rule executes the necessary conceptual model-specific API for retrieving and adding these facts into the knowledge base. It is assumed that this functionality will include a consistency check and an error state if the inclusion of the facts has led to some logical inconsistency in the knowledge base.

5.7.2 Service space API

`execute` abstracts the lower level process of discovering a relevant service, examining it and invoking it.

`execute(srvcDescription, srvcInput, srvcOutput, srvcMessage, srvcResult)`
`srvcDescription` is a string, either having the value "namespace" or "media-type"
`srvcInput` is an URI, either representing the namespace or media-type which represents the intended input to the service
`srvcOutput` is an URI or URIList, either representing the namespace(s) or media-type(s) which represent the permitted output(s) from the service

`srvcMessage` shall be a value which represents what is to be passed to the service
`srvcResult` shall be a variable which is bound to the result of the service execution

The first three parameters form the basis for the discovery of the service, using the guideline described in section 5.4. The service message and result type depends on the service that was executed:

Resource adaptation or conversion ->

The message is a List with an URI pointing to the original resource which is to be adapted or converted, followed by one or more property-value pairs which represent the presentation constraints that the resource must be adapted to meet. If only an URI is passed, then only conversion takes place.
The response is an URL pointing to an adapted and/or converted version of the resource (either stored in a location accessible to the service, or if the resource itself is passed back from the service then the service rule will save that resource in a location accessible to the SWeMPs system)

Knowledge extraction ->

The message is an URL which is the URI of the concept for which knowledge is missing.
The response is an URL pointing to a metadata file which contains the RDF-based information extracted by the service. If the service returns the metadata itself, the service rule saves this metadata and returns the metadata location. If the service returns the knowledge in some other form, an additional resource conversion service can be executed for converting this to RDF/OWL.

Knowledge mapping ->

The message is an URL which is the URI representing a concept for which no matching information has been found in the conceptual model and a List of URIs representing the ontologies currently referenced in the conceptual model.
The response will be a List of URLs which represent URIs that are concepts that are equivalent to the input URI and are drawn from the ontologies used by the metadata in the conceptual model.

If the structure of the parameters passed to the service rule differs from the structures of the parameters in the selected service itself, it is the task of the service planner to handle data re-structuring.

5.7.3 Multimedia model API

`insert` abstracts the lower level process of adding a resource and its properties into the multimedia model. `output` abstracts the lower level process of executing the formatter to translate the multimedia model into a final multimedia presentation.

`insert(rsrc, rsrcType, rsrcProps)`
`rsrc` is an URL which locates a representation of a Resource in the conceptual model
`rsrcType` is a string value, one of “Image”, “Text”, “Animation”, “Audio”, “Video” and “Model”.
`rsrcProps` is a list of property-value pairs

The `insert` rule passes to the multimedia modeller the URL of the Resource, its type and a set of property-value pairs which identify the characteristics of that resource. The type can be determined from the resource metadata, using available mappings (the commonest would be mapping from the MIME type, given in the metadata either as a string or URI). At a minimum, the resource properties need to include the default height and width – if visual, and default duration – if continuous. These pairs consist of an URI identifying the property according to a classification scheme and either an URI (representing a concept from a classification scheme) or a Literal (i.e. a XMLSchema datatype such as a string or integer) identifying the value of the property. The use of URIs permits reasoning over this data, as SWeMPs imposes no single vocabulary for representing resource characteristics/presentation constraints. Rather, Semantic Web reasoning can be employed in conjunction with the use of terms from RDF/OWL ontologies. Values can also be given as Lists, which indicates either – in the case of URIs – a set of complementary values (e.g. a group of font types) where the first possible value should be chosen, or – in the case of Literals – ranges of permissible values (e.g. pairs of values are interpreted as ‘from … to …’). The communicative abstractions are also passed as property-value pairs in which the property URI represents the communicative abstraction and the URI value the other resource with which this resource is constrained by the given communicative abstraction.

When the model is initialised in the rulebase, the insert rule is used to add the presentation constraints which apply to all the resources inserted into the model, e.g. the total height, width and duration of the multimedia presentation. This is done in that the rsrcType is “Model” and the constraints themselves are contained in an URL passed by the method and/or a list of property-value pairs.

`output(model)`
`model` is a variable which takes the URL of the resulting model

In the output rule, we call the method of the multimedia modeller which determines a final form of the multimedia presentation, i.e. allocates valid bound values to all free variables in the multimedia model. The model is then passed to the formatter component and formatted according to the given constraints on output format and saved as a file whose URL is returned to the system through the model variable.

5.8 Conceptual model

In section 4.4, we defined the SWeMPs conceptual model using first the formulism of the chosen ontology development methodology (CLASSIC KR model) and then SHOIQ Description Logic, which is the core logical formulism of the Semantic Web (in OWL-DL). For the implementation, the model is developed using the Protégé authoring tool and the OWL plug-in, in order to produce an ontology which can be expressed using the OWL language, the de facto standard for ontologies on the Semantic Web.

We can check the consistency and coherency of this model once we have modelled the ontology in Protégé. The modelling proves to be a relatively straightforward task as Protégé's KR model is frame-based like CLASSIC. The Protégé ontology editing tool provides an environment to validate our model, visualize it and export it in other formats, which of course includes the Web ontology language OWL.

OWL offers an extended set of logical formalisms which can be applied to the conceptual model. In the context of building the OWL ontology of the conceptual model we made the following design decisions:

- Semantic object is renamed Subject to avoid terminological confusion, as in the context of the Semantic Web every construct in an ontology could be construed as being a semantic object. Likewise, Semantic object metadata is renamed SubjectMetadata.

- URLs are abstracted into the class Occurrence. This is to recognise that in the model some same URLs may be related to multiple resources or services (e.g. when they locate a database which can return different resources) and to support re-use. Furthermore, we differentiate occurrences (URLs pointing to retrievable data) from the URIs used to identify any ontological construct in the model.

- Service description properties handles-media-type and handles-namespace are expanded into from-media-type, to-media-type, from-namespace and to-namespace in order to enable the differentiation of input and output and hence define within the model services which convert media types or map ontologies (as was discussed in 5.4).

- The property has-namespace has as its domain Metadata and Ontology instances, and not generally resources, as the namespace is only relevant in the process for those types of resource.

- Some remaining properties are changed to apply to the domain of Metadata. The property exists-in-domain is applied no longer to the Subject but to the Metadata instance and renamed uses-ontology. The property has-metadata is altered to take Metadata as its domain (like references, which is restricted to the domain of SubjectMetadata and the range of Subject) and renamed to describes for the domain of ResourceMetadata and range of Resource and defines for the domain of ServiceMetadata and range of Service.

- The top level classes Subject, Resource, Service, MediaType, Occurrence and XMLNamespace are declared disjoint, i.e. an individual of one class can not be the member of another class. This is to explicitly state that these concepts are distinct in the model – arguably a resource or service could be a subject, but the semantics of these classes – in the domain of the multimedia generation process – decides that they are to be considered distinct.

The resulting ontology conforms to OWL-DL. That means one can apply DL-based reasoning to this ontology for subsumption testing and consistency checking. It is found that the ontology is valid and consistent. The tests were made by the Protégé tool. The ontology is illustrated in Figure 5.13. Its visualisation was generated by the RDFSViz tool[62].

This conceptual model is an upper level knowledge representation of the domain of multimedia generation. It exists at an abstract, high level where multimedia generation is modelled in terms of the general process that is carried out and the components which exist within that process. For concrete multimedia generation tasks, it is clear that a knowledge base instantiated from this model must also be extended with ontologies specific to the domain of the multimedia generation task. Ontology importing is the act of extending the conceptual model with further knowledge models in its instantiation as a knowledge base. Through ontology importing, instances in the model can also be specified in terms of the extension classes and properties inserted through the imported ontology.

For the multimedia modeller, the ZyX abstract multimedia model [Boll,1999b] was also ontologized and the class structure of this ontology is shown in Figure 5.14.

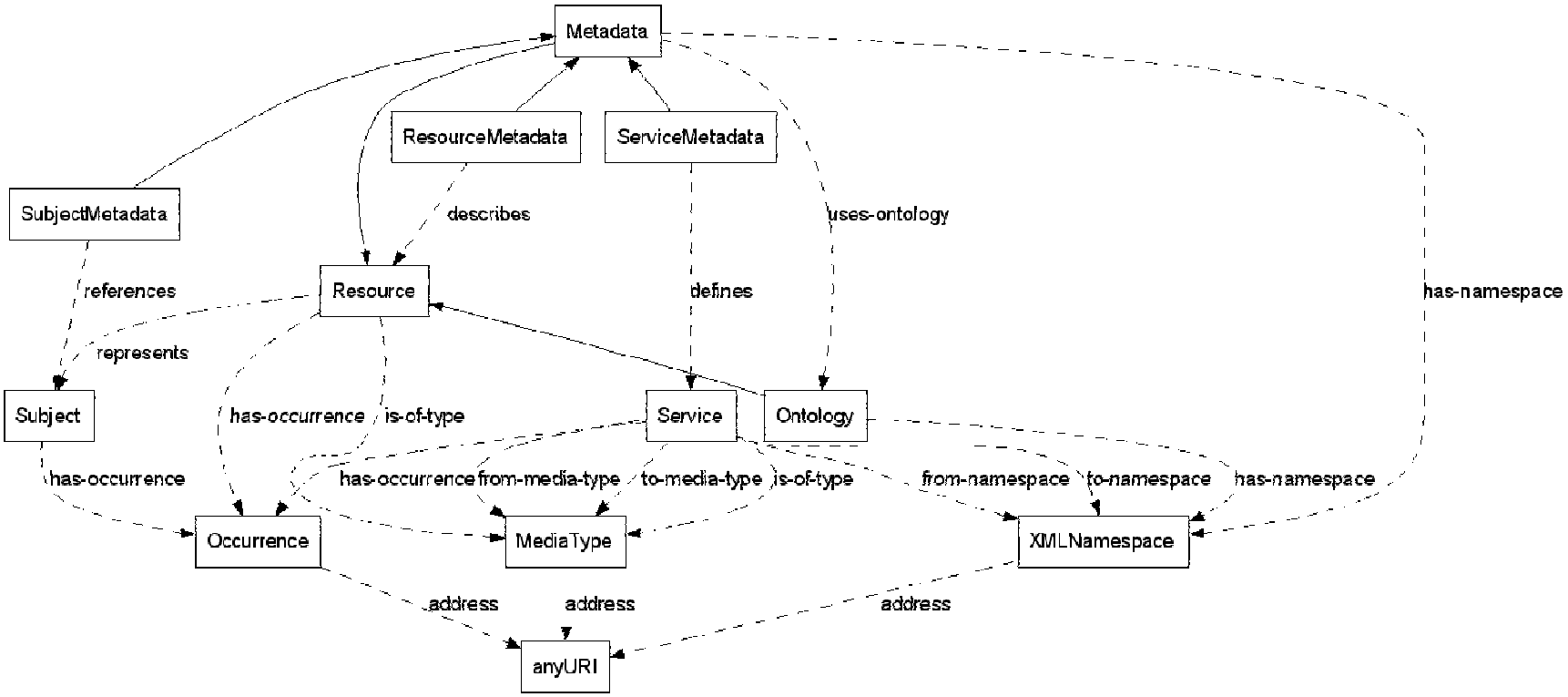

Figure 5.13 SWeMPs Ontology

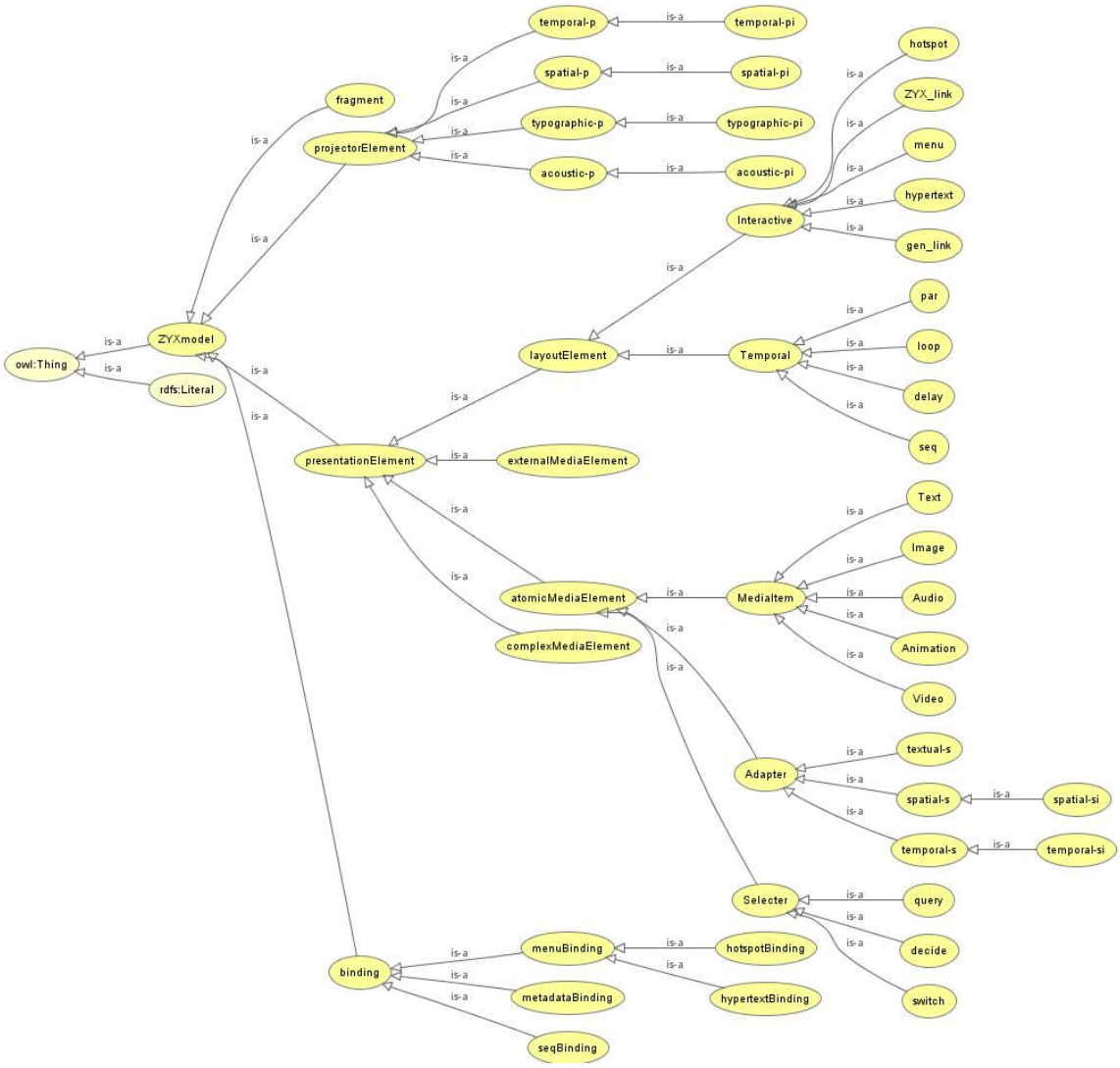

Figure 5.14 ZyX ontology

5.9 Conclusion

In this chapter, we have moved from the proposal of a framework and conceptual model presented in Chapter 4 to a more concrete specification of how the SWeMPs system is implemented. We have followed a known and established software development approach. We outlined which technologies have been considered as best suited for the implementation, defined (abstract) APIs for the components that were defined as part of the architecture and elaborated the rules which model the multimedia generation process using a Prolog-like syntax. Finally we created OWL ontologies to represent the SWeMPs conceptual model and the ZyX abstract multimedia model.