FAST AND ADVANCED STORYBOARD TOOLS

FP7-ICT-2007-1-216048

http://fast.morfeo-project.eu

Deliverable D2.2.3

Ontology and Conceptual Model for the Semantic Characterisation of Complex Gadgets

Knud Möller, NUIG
Ismael Rivera, NUIG
Marcos Reyes Ureña, TID
Ciprian Alexandru Palaghita, Cyntelix

Date: 27/02/2011

FAST is partially funded by the E.C. (grant code: FP7-ICT-2007-1-216048).

Version History

Rev. No.	Date	Author (Partner)	Change description
1.0	27.02.2009	Knud Möller, NUIG	final version (D2.2.1) ready for external review
2.0	27.02.2010	Knud Möller, NUIG	final version (D2.2.2) ready for external review
3.0	27.02.2011	Knud Möller, NUIG	final version (D2.2.3) ready for external review

Executive Summary

This deliverable defines the abstract conceptual model and concrete ontology for the semantic characterisation of complex gadgets in the FAST project. As such, it feeds directly into the various implementation efforts within the project. Throughout the document, we apply the *Methontology* ontology development methodology. Our design and development process is therefore defined by a number of different living documents, which function both as a development tool, as well as documentation for the ontology itself. The major part of the deliverable is made up of these documents, resulting in the final *implementation document* which formally defines the ontology in OWL DL semantics. Throughout the document, we provide extensive code examples and visual representations of the different aspects of both model and ontology.

Document Summary

Code	FP7-ICT-2007-1-216048	**Acronym**	FAST
Full title	Fast and Advanced Storyboard Tools		
URL	http://fast.morfeo-project.eu		
Project officer	Annalisa Bogliolo		

Deliverable	**Number**	D2.2.3	**Name**	Ontology and Conceptual Model for the Semantic Characterisation of Complex Gadgets
Work package	**Number**	2	**Name**	Definition of Conceptual Model

Delivery data	**Due date**	28/02/2011	**Submitted**	27/02/2011
Status			final	
Dissemination Level	Public ☒ / Consortium ☐			
Short description of contents	D2.2.3 is the second iteration of the FAST Gadget ontology. The ontology provides a formal description of the conceptual model of FAST in general, and of the screen/screenflow architecture in particular. The deliverable contains a number of successive intermediate representations which document the development process, and the final product in the form of an OWL ontology. Numerous code examples and visual representations of the ontology terms are provided.			
Authors	Knud Möller, NUIG, Ismael Rivera, NUIG, Marcos Reyes Ureña, TID, Ciprian Alexandru Palaghita, Cyntelix			
Deliverable Owner (Partner)	Knud Möller, NUIG		**email**	knud.moeller@deri.org
			phone	+353 91 495086
Keywords	FAST, conceptual model, ontology, OWL, RDFS			

Table of contents

Lists of Tables and Figures

List of Tables

List of Figures

Listings

1 Introduction

1.1 Goal and Scope

As the corner stone of the theoretical work undertaken in the FAST project, the definition of a conceptual model and ontology for the characterisation of complex gadgets feeds directly into the project's three main development activities: the development environment for complex gadgets (GVS), the development of gadget components and the semantic catalogue which represents the backend of the architecture. For all these different aspects of FAST, the ontology developed and presented in this deliverable provides a structured definition of their common domain of discourse. The ontology formally defines the parts which a complex gadget is made of, how the parts interrelate, what kind of metadata is available for each part, how users of the system are represented, etc. It should be pointed out that the conceptual model and ontology are not meant to cover every conceivable aspect of the development of complex gadgets, not even every conceivable aspect of the FAST platform. Rather, the goal is the following: FAST is about the *development of complex gadgets for end users*, and so the conceptual model and ontology provide the terminology and conceptual framework that is needed for specifically this task. In other words, while this document provides the theoretical underpinning for the implementation-oriented work in the project, the implementation-oriented work on the other hand defines the scope and requirements for the model.

While following an ontology design methodology which is agnostic to particular data models or implementation languages (see Appendix A), we have chosen to represent the final ontology specification using the Resource Description Framework (RDF) and OWL semantics.

In FAST, we have adopted the term *gadget* for the small, Web-based software components which the project focusses on. There are small differences in meaning between this term and the term *widget*, which is also widely used. However, for the purpose of this document, it should be noted that whenever we use the term *gadget*, we might also have used the *widget* at the same time.

1.2 Structure of the Document

This deliverable is structured as follows: we will start off by outlining the main changes that this document has undergone since its previous version, so as to enable the reader to quickly identify new material. After briefly addressing the topic of related work in Sect. 2 — which is mainly covered in a different deliverable from this work package, [Urmetzer et al., 2010] — the structure of the following sections follows directly from our adopted ontology development methodology: we will perform a domain analysis in Sect. 3, covering the stages of specification, conceptualisation and integration of existing vocabularies. This is then followed by the implementation step in Sect. 4, which covers the concrete ontology in its form at this moment of the project, as well as some basic considerations such as an early proposal for prototypical semantics for RDF. After a section discussing an evaluation of the FAST gadget ontology in Sect. 5, the deliverable will be concluded with a summary of our work in Sect. 6.A.1 A number of appendices have been added to the document: App. A presents the development methodology adopted by us, App. B provides documentation for the ontology in the form of a detailed list of terms, while App. C contains the complete code of the ontology.

1.3 Changes from Previous Version

The following changes have been made with respect to the previous version of this deliverable, D2.2.2. [Möller, 2010]. A history of previous changes is kept in App. E.

- Section 4.2 has been significantly extended with a more in-depth discussion of prototype theory, and a first proposal for prototype-based semantics for RDF, as they are employed in FAST.
- A section for the *VocDoc* tool, which was developed in part for generating online and PDF documentation for the FAST ontology, has been added as App. D.
- The ontology itself has remained more or less stable since D2.2.2. [Möller, 2010]. However, a handful of small changes and corrections have been performed, e.g., to reflect the terminology changes in the prototype section (*prototype* instead of *template*, *clone* instead of *copy*).

- The document as a whole has been revised, and minor changes (mostly, spelling, grammar and small updates to reflect latest developments) have been throughout the deliverable.
- The online documentation of the FAST Gadget ontology at `http://purl.oclc.org/fast/ontology/gadget` has been updated to reflect the changes in this version of the deliverable.

2 Related Work

We are not aware of any other project which has already developed or planned to develop an ontology of the gadget domain. However, there is a range of related material from which we take inspiration regarding the description of the gadget domain. This material comes from a number of different sources and directions. It includes work done under the hood of the W3C which may eventually lead to an official specification of various technical aspects of the gadget domain, as well as a number of different gadget APIs. While both are not strictly speaking ontologies, they nevertheless provide a very good insight in how to conceptualise the domain. Additionally, we consider work done in the area of Semantic Web Services, firstly because the gadgets designed in FAST will eventually connect to Semantic and conventional Web services, and secondly because there are a number of similarities between the way SWS are often formalised and the way we envision the interaction of the different components of a gadget in the FAST IDE.

All of these references are discussed in [Urmetzer et al., 2010], and we refer the reader to this deliverable for more detail.

3 Domain Analysis and Conceptualisation

In this section, we will apply a number of the phases proposed in Methontology (see Sect. A.1) to the process of developing the FAST gadget ontology. The names of the following sections reflect the names of the phases in Methontology.

3.1 Specification

The specification phase of Methontology requires setting up an ontology requirements specification document. We have compiled such a document for the FAST Gadget Ontology, as in Tab 1.

Table 1: FAST Ontology Requirements Specification Document

Name	**FAST Gadget Ontology**
Domain	Intelligent and complex gadgets
Purpose	The FAST gadget ontology conceptualises the domain of intelligent gadgets as defined in the FAST development platform. Gadgets consist of several interrelated parts, all of which are covered in the ontology. In FAST, a description of each gadget component and resource is available to the IDE. From this description, the IDE can construct its interface, determine which components can be connected to which other components, or make suggestions to the user in order to aid in the gadget development process. Furthermore, FAST gadgets are capable of connecting to a variety of backend Web services, which can either be Semantic Web services, or conventional, non-semantic Web services. Therefore, the FAST Gadget Ontology must facilitate the description of those backend services as well. Where a semantic description of the service already exists, it may be necessary to perform ontology mediation between the ontology of the backend service description and the FAST ontology. Finally, the FAST IDE will support individual user profiles to allow personalisation of the gadget development process. This means that the FAST gadget ontology will also have to cover the description of users and user profiles. In summary, the ontology is supposed to facilitate support for the user in the design and generation of FAST gadgets, as well as in searching and browsing those gadgets.
Level of Formality	formal (OWL ontology)
Scope	**Concepts:** *Component*, *Resource*, *Screen*, *Screen flow*, *Backend Service*, *Flow Control Element*, *Operator*, *Screenflow Start*, *Screenflow End*, *Connector*, *Form Element*, *Pre-condition*, *Post-condition*, *Query*, *Label*, *Icon*, *User*, *User Profile*, *Tag* **Properties:** *containsScreen*, *hasLabel*, *hasIcon*, *hasPreCondition*, *hasPostCondition*, *hasTag*, *hasProfile*

Sources of Knowledge	FAST Architecture deliverable [Ureña and Solero, 2010], FAST Requirements Specification [Solero et al., 2010], FAST State-of-the-art Document [Urmetzer et al., 2010], EzWeb documentation [Lizcano et al., 2008]

3.2 Conceptual Model

In the conceptualisation phase, we first produce a *glossary of terms* which lists all classes and properties of our ontology. The glossary of terms is seeded from the specification document (see previous section), but groups the terms according to relatedness. Also, the evolution of the ontology is mainly reflected in this document, as new terms being introduced based on the employment and ongoing evaluation of the ontology are added here, whereas the specification document remains in its original state. After introducing the general terms, the conceptual model is then refined by relating the different terms with regards to type hierarchy, composition and aggregation.

3.2.1 Glossary of Terms: Classes

"Classes" here should be read as "types of things" in general. I.e., if a term is listed as a *class* here, this has no direct implications on the concrete implementation of the term. For example, while *label* is a type of thing that is important in FAST, it will not necessarily be represented as an `owl:Class` in the implementation step, but could just as well be represented as a property. Table 2 lists all classes considered in the conceptual model of FAST, loosely grouped according to their relevance for specific aspects of the platform.

Table 2: FAST Glossary of Terms, Classes

Name	Description
Gadgets, Components and Building Blocks	
Gadget	A FAST gadget is the wrapped and deployed instance of a Screen_Flow. It is the end result of the FAST workflow. As such, it is an important part of the conceptual model of FAST. However, it will not be modelled as an entity in the FAST ontology, since it does not does need to be handled by either GVS or catalogue.
Building_Block	Anything that is part of a gadget. Tentatively anything that can be "touched" and moved around in the FAST IDE, from the most complex units such as Screen_Flows, down to atomic Form_Elements like a button or a label in a form.
Screen_Flow	A set of screens from which a gadget in a given format (which can cover one or more target platforms) can be generated.
Screen	An individual screen; the basic unit of user interaction in FAST. A screen is the interface through which a user gets access to data and functionality of a backend service.
Screen_Component	Screens are made up of screen components, which fundamentally include service Resources, Operators and Forms.
Resource	A service resource in FAST is a wrapper around a Web service (the BackendService), which makes the service available to the platform, e.g., by mapping its definition to FAST facts and actions. Note that this meaning of the term "resource" is different from the previous version of the FAST ontology in [Möller, 2009].
Backend_Service	A Web service which provides data and/or functionality to a screen. The actual backend service is external to FAST, and only available through a wrapper (the service Resource).
Operators	Operators are intended to transform and/or modify data within a screen, usually for preparing data coming from service resources for the use in the screen's interface. Operators cover different kinds of data manipulations, from simple aggregation to mediating data with incompatible schemas.
Form	A form is the visual aspect of a screen: its user interface. Each form is made up of individual form elements.
Form_Element	Form elements are UI elements in a particular Screen, such as buttons, lists or labels.
Pipe	Pipes are used to explicitly define the flow of data within a screen, e.g., from service resource to operator to a specific form element.
Action	Actions are representations of some specific functionality of a building block in FAST. Examples are methods of a Web service (e.g., getItem) or functionality to update or change the contents of a Form.
Trigger	Triggers are the flip-side of actions. Certain events in a building block can cause a trigger to be fired. Other building blocks within the same screen, which are listening to it, will react with an Action.
Pre- and Post-conditions	

Name	Description
Condition	The pre- or post-condition of a certain kinds of building block. If the building block is a Screen_Flow, each target platform will use these conditions in its own way, or may also ignore them. E.g., in EzWeb pre- and post-conditions correspond to the concepts of *slot* and *event*.
Pre-condition	The pre-condition of a building block comprises of a set of Facts that need to be fulfilled or available in order for the building block to be activated.
Post-condition	The post-condition of a building block is a set of Facts it can produce, and which will be thus become available to other building blocks.
Pattern	A set of facts which formally defines a condition. A pattern may contain one or more variables.
Fact	The "basic information unit of a FAST gadget" [Ureña and Solero, 2010]. In terms of RDF, a fact is a statement consisting of a subject, predicate and object *(S, P, O)*.
Ontology	Facts in FAST are represented as RDF triples, using terminology from a particular vocabulary or ontology. A building block can explicitly specify which ontology or ontologies it uses to specify its pre- and post-conditions (e.g., in the case of a Screen), or which ontology or ontologies it can otherwise handle (e.g., in terms of a mediation Operator).
Implementation	
Code	In case the implementation of a building block is hard-coded, the Code represents the source code that defines it.
Library	For hard-coded building blocks, certain programming libraries may be necessary. The are represented as instances of Library.
Definition	For building blocks which do not rely on a hard-coded implementation, but which are instead defined declaratively, the definition is represented by this class.
User and User Profiles	
User	A human user involved in the lifecycle of a FAST gadget. The kind of User in FAST that is most relevant to the gadget ontology is a user of the FAST IDE, since these users need to be represented and handled by the FAST platform directly (as opposed to gadget end users).
User_Profile	The settings and data known about a particular user of the FAST platform.
User_Role	Throughout the lifecycle of a FAST gadget different kinds of users are involved. These users are distinguished by playing a particular User_Role. The particular user roles identified in [Solero et al., 2010] are *end users* of a gadget, *key users*, *consultants* (i.e., gadget developers), *screen developers* and *resource developers*. Any particular User can theoretically play several roles at the same time.
Annotation	
String	A catch-all class of objects that serves for representing short labels, longer descriptions, dates, patterns, etc. If meant for human consumption, strings can have multiple representations for different languages.

Name	Description
Image	Images are any kind of graphical object, such as icons, screenshots, etc., and are used in the UI of the FAST platform to represent building blocks to users.
Tag	A short (usually one word) description of a building block. Rather than being only simple literal string, a Tag in FAST is a complex object consisting of a string representation as well as a unique identifier, which unambiguously specifies the tags meaning.

3.2.2 Glossary of Terms: Properties

All relevant properties of the FAST conceptual model are listed in Tab. 3. Not all properties which will eventually be defined in the formal ontology are listed in the table: in many cases, these concrete properties simply reflect particular instances of generic composition or aggregation relations, which become apparent in the diagrams in Sect. 3.2.3. Usually, these concrete properties will follow a general naming scheme of hasCLASS_NAME, e.g., hasDefinition or hasPreCondition.

Table 3: FAST Glossary of Terms, Properties

Name	Description
Components/Building Blocks	
contains	Many kinds of building blocks in FAST can contain other building blocks: screenflows contain screens, screens contain forms or operators, forms contain form elements, etc. In many cases, specific properties for particular containment relations could be specified in the formal ontology.
hasPreCondition	This property links a building block to its pre-condition, i.e., the facts that need to be fulfilled in order for this screen or screenflow to be reachable.
hasPostCondition	This property links a screen or screenflow to its post-condition, i.e., the facts that are produced once the screen or screenflow has been executed.
Pre- and Post-conditions	
hasPattern	This property links a condition resource to the pattern which formally defines it.
isPositive	Conditions can be positive or negative, depending on whether they must be fulfilled or must not be fulfilled (in the case of pre-conditions), or whether their facts will be added to the canvas or removed (in the case of post-conditions).
Annotation	
hasLabel	A string attached to any FAST component or sub-component, which represents a short (∼1-2 word) human-readable description of the component.

Name	Description
hasDescription	A string attached to any FAST component or sub-component, which represents a longer, more detailed human-readable description of the component in natural language.
hasImage	An abstract property defining the relation between a thing in FAST and an image which somehow represents it.
hasIcon	A small graphical representation of any FAST component or sub-component.
hasScreenshot	An image which shows a particular screen or screenflow in action, to aid users in deciding which screen or screenflow to choose out of many.
hasRights	A human-readable description of license information or similar for a screen, screenflow or possibly other components.
hasCreator	Every resource generated within FAST will contain metainformation about the person who created it.
hasVersion	A string representing the version number of a resource in FAST.
hasCreationDate	A date-formatted string representing the date when a resource in FAST was created first.
hasHomepage	Links a resource in FAST to a main Webpage with information about it. We expect screenflows/gadgets to have homepages, as well as users. Other resources will most likely not have homepages.
hasDomainContext	A catch-all property to annotate a resource in FAST with information about the domain for which it is relevant. The domain context will be utilised for searching, matching, etc. Domain contexts can be expressed either as tags or structured objects.
User and User Profiles	
hasName	The full name of a FAST user.
hasUserName	The user name of a FAST user, which will often be a short form of the full name, or a nick name. The user name must be a unique ID within an instance of FAST.
hasEmail	The e-mail address of a FAST user.
hasProfile	Connecting a user with their profile.
hasInterests	Interests of a user as specified in their profile. Interests could be matched with a component's domain context to allow the FAST catalogue (and thereby the GVS) to suggest screens and other components to the user.

3.2.3 Term Relations within the Conceptual Model

Regarding the layering from the actual gadget down to the embedded Web services, the central classes in the conceptual model of FAST can be grouped into three levels: the gadget and screen flow level at the top, the level of individual screens in the middle, and the level of Web services at the bottom. This layering is reflected in Fig. 1, which illustrates the central compositional relation-

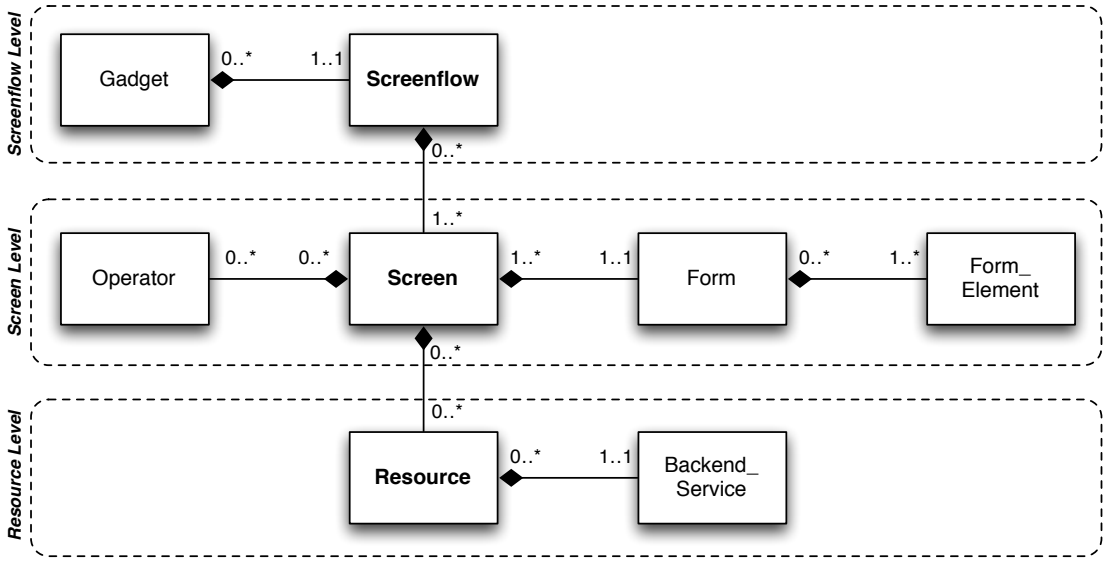


Figure 1: Important composition relationships within the FAST conceptual model

ships of the model. As the figure shows, a *gadget* in FAST is mainly composed of a *screen flow*, which it wraps. The *screen flow* is composed of one or more *screens*, which is turn comprises the three *screen components* of the visual front-end (the *form*), the back end service (the *resource*) and the data *operator*. Finally, *forms* are assembled from *form elements*, while *resources* wrap *backend services*.

Figure 2 looks closer at the features of individual screens. These are forms, resources and operators, which are all specialisations of the general *screen component* concept. Each such screen component may or may not have an action or trigger associated with them, which declaratively define their behaviour in terms of their own functionality or functionality they can trigger in other building blocks.

Regarding their implementation, different kinds of building blocks in FAST can either be defined hard-coded as a piece of source code, or they can be defined declaratively in the terms of the FAST ontology (using pipes, operators, etc.). In the former case, a building block such as a screen would have been implemented and added to the FAST platform by an engineer, whereas in the latter case, ordinary users of the FAST platform would have assembled the building block using the tools available in the GVS[1]. This principle is reflected in Fig. 3, showing how certain kinds of building blocks aggregate either a `Definition` or `Code`, whereas other kinds of building blocks are always hard-coded. It should be noted that, in theory, there is no reason why not any kind

[1]At this moment, only screens can be assembled in the GVS. Forms may or may not be editable in the GVS in the future. Other kinds of building blocks are not currently planned to be editable.

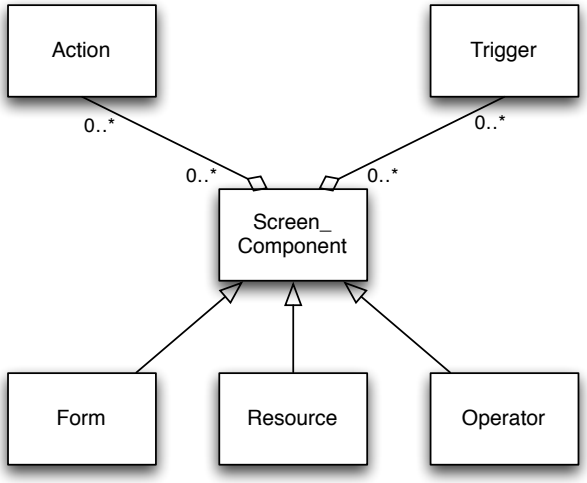


Figure 2: Aggregation of screen components

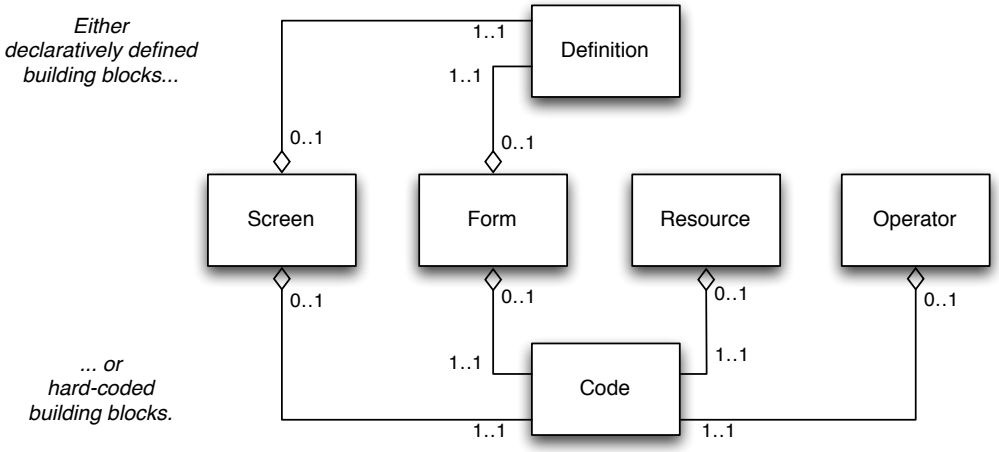


Figure 3: Building blocks have either code or are defined declaratively

of building block could be defined either way. However, the concrete set of building blocks which can be defined declaratively or in code is changing dynamically according to the current state and plans of the FAST project. The situation depicted in Fig. 3 — `Screen` and any kind of `Screen_Component` can be defined in code, whereas only `Screen` and `Form` can be defined declaratively — reflects the status at M24 of the project.

Figure 4 illustrates how different building blocks relate to the concepts of pre- and post-conditions. While all building blocks can essentially have such conditions, only screens and screen flows directly aggregate both kinds of conditions. However, in the case of the three different kinds of screen components (form, resource and operator), the relation to a pre-condition is only estab-

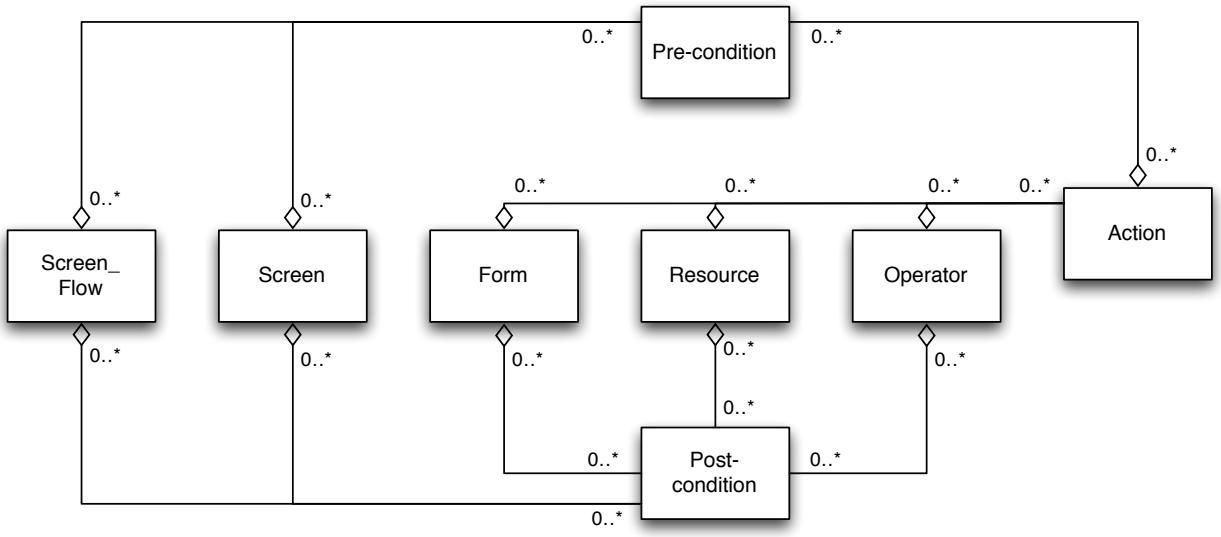


Figure 4: Building blocks with pre- and post-conditions

lished through their actions, which represent their basic functionality. For all building blocks, both pre- and post-conditions are entirely optional.

3.3 Integration

In this section we will investigate a number of established ontologies which cover part of the requirements laid out in the specification document, as well as the glossary of terms from the conceptualisation stage. In particular, we will look at the Dublin Core metadata specification, the Friend of a Friend (FOAF) ontology, Semantically Interlinked Online Communities (SIOC) and Common Tag. The outcome of the integration process is the living *integration document* at 3.3.5.

3.3.1 Dublin Core

Dublin Core (DC) is a set of metadata elements for describing online resources in order to support indexing, searching and finding them. Typical examples of terms from DC are *title*, *creator*, *subject*, *description*, *date* or *rights*. In principle, DC can be applied to a wide range of resources, but is typically used in the context of describing media (textual documents, video, sound or image).

The initial work on DC was done at an invitational workshop in Dublin, Ohio, US (hence the name) in 1995. Since then, several revisions and extensions have been applied to the vocabulary.

While it is possible to express DC in plain HTML, the more widely used language is probably RDF. The original 15 terms were all properties defined in the DC elements namespace (`http://purl.org/dc/elements/1.1/`, short `dc`). The properties were not formally defined with respect to their range and domain, or whether they are object or datatype properties. However, the assumption was that the values for each property would always be literals, as opposed to complex values represented by a URI. More recently, the DC elements namespace has been declared legacy and is now superseded by the DC terms namespace (`http://purl.org/dc/terms/`, short `dcterms`) [DCMI Usage Board, 2008], which includes more precise re-definitions of all 15 original terms, as well as a number of new terms (resulting in a total of 55 terms). This new iteration of DC now provides the facility to use structured values for properties and specifies what type these values will have. E.g., the property `dcterms:creator` now supersedes the original `dc:creator` and specifies that its value has the type `dcterms:Agent`.

In order to reconcile with the simplicity of the original, flat DC elements and the newer, more structured DC terms, DC also includes the so-called “DumbDown” principle, which basically says that structured values of DC terms should always carry a literal description as well, in order to cater for “dumb” agents processing the data. These literals should be expressed using properties such as *rdfs:label* or *rdf:value*.

Because of the open nature of the RDF model and the open-world assumption usually applied for RDFS and OWL, it is possible to integrate DC with other ontologies such as FOAF. E.g., it is possible and good practice to make a statement saying that the `dcterms:creator` of a document is an instance of `foaf:Person`. By inference following from the DC terms specification, this person would then also be a `dcterms:Agent`. This scenario is illustrated in Fig. 5. A similar, but slightly out-dated example is given in `http://dublincore.org/documents/dcq-rdf-xml/`.

Looking at the requirements specification document, DC can be integrated into the FAST gadget ontology in many ways. A lot of the metadata needs for screens, screenflows and their components are identical to those of the media resources originally intended by DC. For this reason, many properties such as `title`, `creator`, `rights` or various `date` properties can be used directly (more details in the integration document).

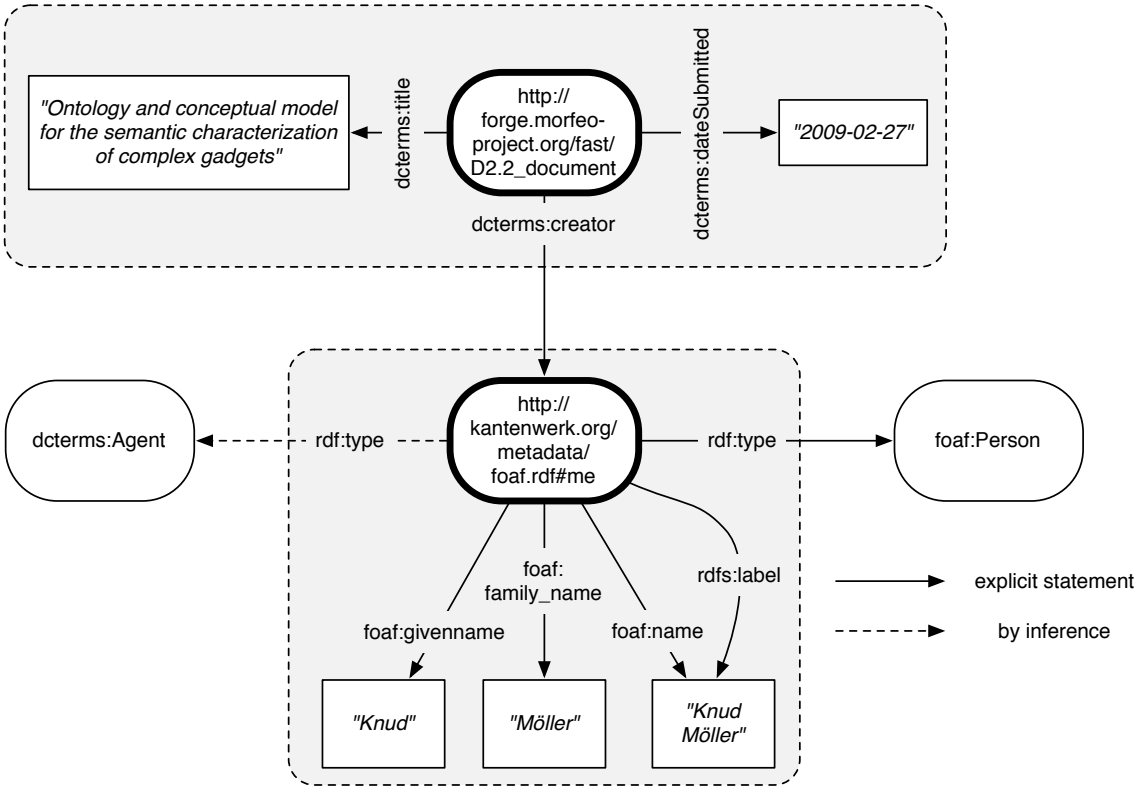


Figure 5: Example of Dublin Core data

3.3.2 FOAF

The Friend of a Friend vocabulary is a simple ontology which main purpose to describe people — represented as instances of the `foaf:Person` class — in terms of their contact information, interests, or Web presence. More importantly, however, is the fact that FOAF provides some simple terms for specifying who knows who (using the propery `foaf:knows`), thereby effectively allowing to build a social network of people. There is no such thing as a centralised FOAF service to which people have to sign up. Instead, each person can maintain and host their own FOAF description (often in the form of a *FOAF file*), or use one of many external services which provide this functionality, such as LiveJournal, TypePad, Vox, Hi5, etc., thus making the FOAF network a truly decentralised social network. Figure 6 depicts an excerpt of some typical FOAF files, describing two people called “Knud Möller” and “Andreas Harth”, including the fact that Knud knows Andreas.

FOAF has been *“evolving gradually since its creation in mid-2000”* [Brickley and Miller, 2007]. Over the years, it attracted a lot of attention and has become one of the major successes of the

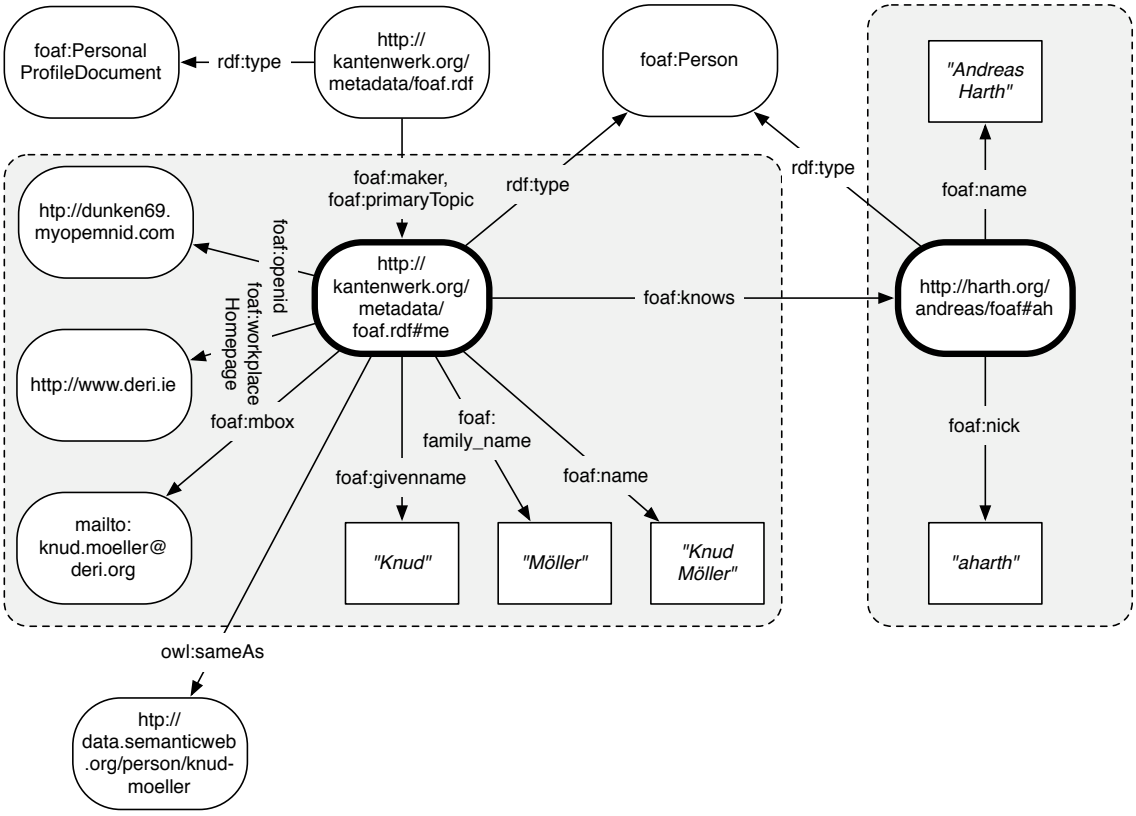


Figure 6: Example of FOAF data

Semantic Web. Compared to most other ontologies or vocabularies, it has received a much higher uptake, with the number of FOAF files on the internet probably numbering tens of millions now. Its popularity in the community and beyond is also underlined by the fact that many other ontologies are integrating FOAF in order to represent basic information regarding people, or they are using it as a point of reference and extend the basic FOAF classes and properties for their own needs. Examples for this kind of integration are the increasingly popular SIOC (Semantically-Interlinked Online Communities) ontology [Breslin et al., 2005], which uses FOAF description to unify identities in different online communities, or the Semantic Web Conference ontology [Möller et al., 2007], where conference attendees or authors are represented as FOAF person instances.

The specification document of the FAST ontology defines that part of the purpose of the ontology is “the description of users and user profiles”. The most obvious solution for implementing this requirement is to integrate the FAST ontology with FOAF. This means that each user of FAST would be modelled as a `foaf:Person`. A lot of the basic necessities in terms of user profile information is covered by properties defined in FOAF (names, contact details, interests, user pictures, etc.) and could therefore be used as is. In other cases, rather generic FOAF terms

might be good starting points for extension towards more specific needs that occur in FAST. The icons of various components in the GVS may serve as an example here: the relation that holds between an icon and the thing X of which it is an icon could be described as “the icon depicts X”. FOAF provides the property `depiction` to express this general relation. However, `hasIcon` (our property for saying that something is the icon of a thing) can be considered a specialisation of this, and so we will say that `fgo:hasIcon` is a specialisation of `foaf:depiction`. In terms of RDFS, we will say that `<fgo:hasIcon> <rdfs:subPropertyOf> <foaf:depiction>`.

3.3.3 SIOC

Another vocabulary that is gaining a lot of uptake and support on the Semantic Web is SIOC, which received additional weight when it became a W3C member submission in 2007. The basic idea behind SIOC is that there is an abstract model behind all online community sites which contain an aspect of discussion between members, be it forum sites, discussion boards, blogs, wikis, content management systems, mailing lists, etc. For each of those *sites*, it can be said they contain discussion threads or *fora*, which in turn contain *posts*, which in turn can have *comments*. Furthermore, each post or comment can have a *topic* and a *creator*, which can be a *user* of the site. The authors of SIOC make a point of integrating the vocabulary with FOAF, e.g., by suggesting that a user of a site is in fact held by an instance of `foaf:Person` (the SIOC specification states that `sioc:User` is a sub-class of `foaf:OnlineAccount` [Breslin and Bojārs, 2009]). An overview of the different classes and properties of SIOC is given in Fig. 7.

From an implementation point of view, a particular site will use SIOC by exposing its internal data on request with the help of a SIOC exporter. Such components have been provided by various members in the SIOC developer community for different popular platforms, such as the WordPress blogging software, the Drupal CMS or the Twitter micro-blogging platform. The benefit that sites are creating for end users by using SIOC is that a common representation format and reference points such as authors and topics allow data from different sites to be integrated and thus browsed and searched together.

From the point of view of the FAST gadget ontology, the most interesting features of the vocabulary are the classes and properties it provides for modelling users of a site. We will use SIOC in combination with FOAF to model users and their profiles, as well as user groups, as specified

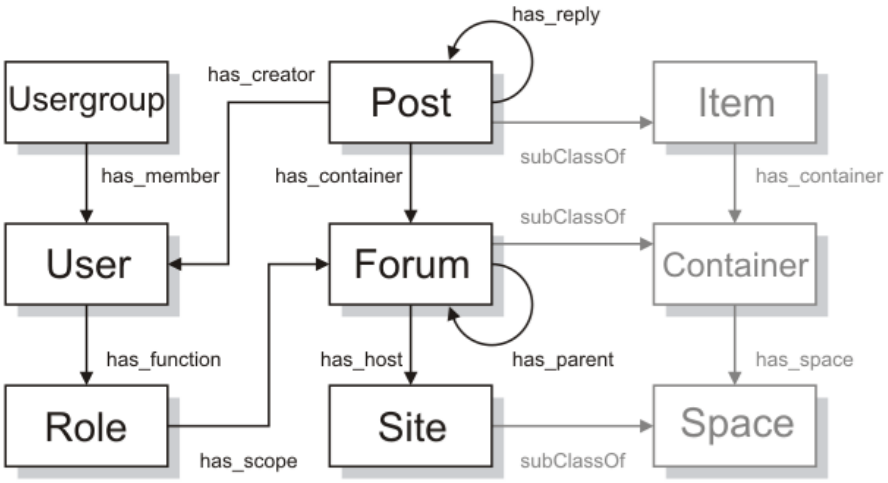


Figure 7: Overview of the SIOC ontology

in the requirements document. The discussion aspects of SIOC will not have an immediate use within the FAST platform. However, it is conceivable that collaborative features such as support for fora will be added to FAST at some point, which will lead to an extension of the requirements document. These additional requirements could then be fulfilled by the integration of further terms from SIOC.

3.3.4 Common Tag

Common Tag[2] is an open format and initiative for conceptual tagging, which is developed and backed by a number of relevant players in the Web community, including Yahoo! and DERI. The basic idea of conceptual tagging is to address the problem of ambiguity in the meaning of tags (e.g., does "jaguar" mean the car, the animal or the operating system?) as it occurs in free-text tagging. Rather than relying on methods such as clustering, applied to a whole corpus of tagged resources, conceptual tagging intends to disambiguate a tag right from the moment it is used, by linking it to an unambiguous concept from vocabularies such as Dbpedia [Auer et al., 2007] (representing Wikipedia as linked data) or Freebase[3]. E.g., in order to disambiguate "cocoa" as meaning the API, not the drink or plant, it would be represented as a resource (rather than a plain literal) and linked to the DBpedia resource `http://dbpedia.org/resource/Cocoa_`

[2] `http://www.commontag.org`, 24/01/2010
[3] `http://www.freebase.com/`

```
@prefix ctag: <http://commontag.org/ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://confuseddevelopment.blogspot.com/2006/03/embeddable-frameworks-in-cocoa.html>
    ctag:tagged _:x .

_:x a ctag:Tag;
  ctag:label "cocoa";
  ctag:means <http://dbpedia.org/resource/Cocoa_%28API%29> ;
  ctag:taggingDate "2010-01-20"^^xsd:dateTime.
```

Listing 1: Common Tag example — disambiguating the tag “cocoa”

`%28API%29`.

Apart from facilitating the disambiguation of tags, Common Tag also allows to make further assertions about the nature of a tag, such as who applied it or when it was applied. As an example, List. 1 shows how a blog post has been tagged on 20/01/2010 with “cocoa”, meaning the API, not the drink or plant.

Within FAST, Common Tag is used to cover the tagging needs as specified in the ontology requirements specification, and in particular in order to define the domain context as specified in the annotation part of the glossary of terms (`Tag` and `hasDomainContext`).

3.3.5 Integration Document

This living document specifies how the terms from the glossary of terms are expressed using terms from the various ontologies presented in the previous sections.

Table 4: FAST Ontology integration document

FAST Term	Integrated Ontology	Integrated Term	Comment
		Classes	
`User`	FOAF	`foaf:Person`	A user in FAST will be modelled using a combination of FOAF and SIOC, so that an instance of `foaf:Person` will have an account on the FAST platform, represented by an instance of `sioc:User`, which is a subclass of `foaf:OnlineAccount`. The relationship is expressed using `foaf:holdsAccount`. This way of modelling directly follows the suggestions made by [Breslin et al., 2007].
`User`	SIOC	`sioc:User`	s.a.
`Image`	FOAF	`foaf:Image`	—
`Tag`	Common Tag	`ctag:Tag`	Complex or conceptual tags are being used e.g. to model the domain context of resources.
		Properties	
`hasLabel`	DC	`dcterms:title`	—
`hasCreator`	DC	`dcterms:creator`	DC defines the range of `dcterms:creator` to be `dcterms:Agent`. By inference, this obviously also holds within FAST. However, we will explicitly only use `foaf:Person` (also see Fig. 5).
`hasDescription`	DC	`dcterms:description`	—
`hasCreationDate`	DC	`dcterms:created`	Dates should be formatted according to ISO 8601 [Wolf and Wicksteed, 1997].
`hasRights`	DC	`dcterms:rights`	DC terms specify a number of classes to model the rights of resources. `dcterms:rights` links a resource to a `dcterms:RightsStatement`. The holder of the rights is specified by pointing from the resource to a `dcterms:Agent`, using the `dcterms:rightsHolder` property.
`hasImage`	FOAF	`foaf:depiction`	The inverse `foaf:depicts` could also be used.
`hasIcon`	FOAF	sub-property of `foaf:depiction`	Icons are a specific kind of image, so we can relate this property to the `foaf:depiciton`.
`hasScreenshot`	FOAF	sub-property of `foaf:depiction`	Screenshots are a specific kind of image, so we can relate this property to the `foaf:depiciton`.

hasName	FOAF	foaf:name	Apart from foaf:name, which is used for the full name, more fine-grained properties such as foaf:firstName, foaf:surname or foaf:title and foaf:nick can be used as well.
hasEmail	FOAF	foaf:mbox	While foaf:mbox will be used for the plain e-mail address, foaf:mbox_sha1sum will be used for an obfuscated version of the address.
hasInterests	FOAF	foaf:interest	—
hasUserName	FOAF	foaf:accountName	—
hasDomainContext	Common Tag	ctag:tagged	The domain context of a resource in FAST can be expressed as a complex tag. Further Common Tag properties can then be applied to make assertions about the tag itself.
hasVersion	DOAP	doap:revision	DOAP (Description of a Project) is a widely used vocabulary for basic annotation of software and other projects.

4 The FAST Gadget Ontology

While the previous section discussed the various steps preceding the definition of the actual ontology according to our chosen development methodology, this section focusses on concrete usage of the FAST gadget ontology in an RDFS/OWL environment. As a point of reference, an overview of all concrete classes will be given in Sect. 4.1, followed by a number of sections detailing concrete aspects of the usage of the FAST gadget ontology.

4.1 Ontology Overview

This section gives a general, birds-eye-view of the ontology in terms of its classes and properties. Fig. 8 shows all classes in their subsumption hierarchy as a UML diagram, including the properties defining those classes as their domain. All `rdfs:subClassOf`/`rdfs:superClassOf` relations are modelled with the subsumption arrow, while `owl:union_of` relations are modelled using the filled composition diamond. A more detailed list of classes and properties, complete with human-readable comments and documentation can be found in the back of the document in App. B, while the complete ontology code in OWL is given in App. C.

4.2 Levels of Representation

In FAST, resources (gadgets, screenflows, screens and their components) exist in different stages throughout their lifecycle. Moving top-down from the high-level conceptual description to gadgets in use at runtime, the following differences can be made in terms of the level of instantiation:

- **Abstract Class Level**: At this level, resources are defined on a categorical level, differentiating between the various kinds of things that are relevant in FAST, as fundamental classes such as screens, forms or resources (see Sect. 3.2). These are implemented technically as classes in OWL.
- **Prototypical Component Definition**: When a user of the FAST GVS is building a new screenflow or screen, they do so by combining existing building blocks and configuring them.

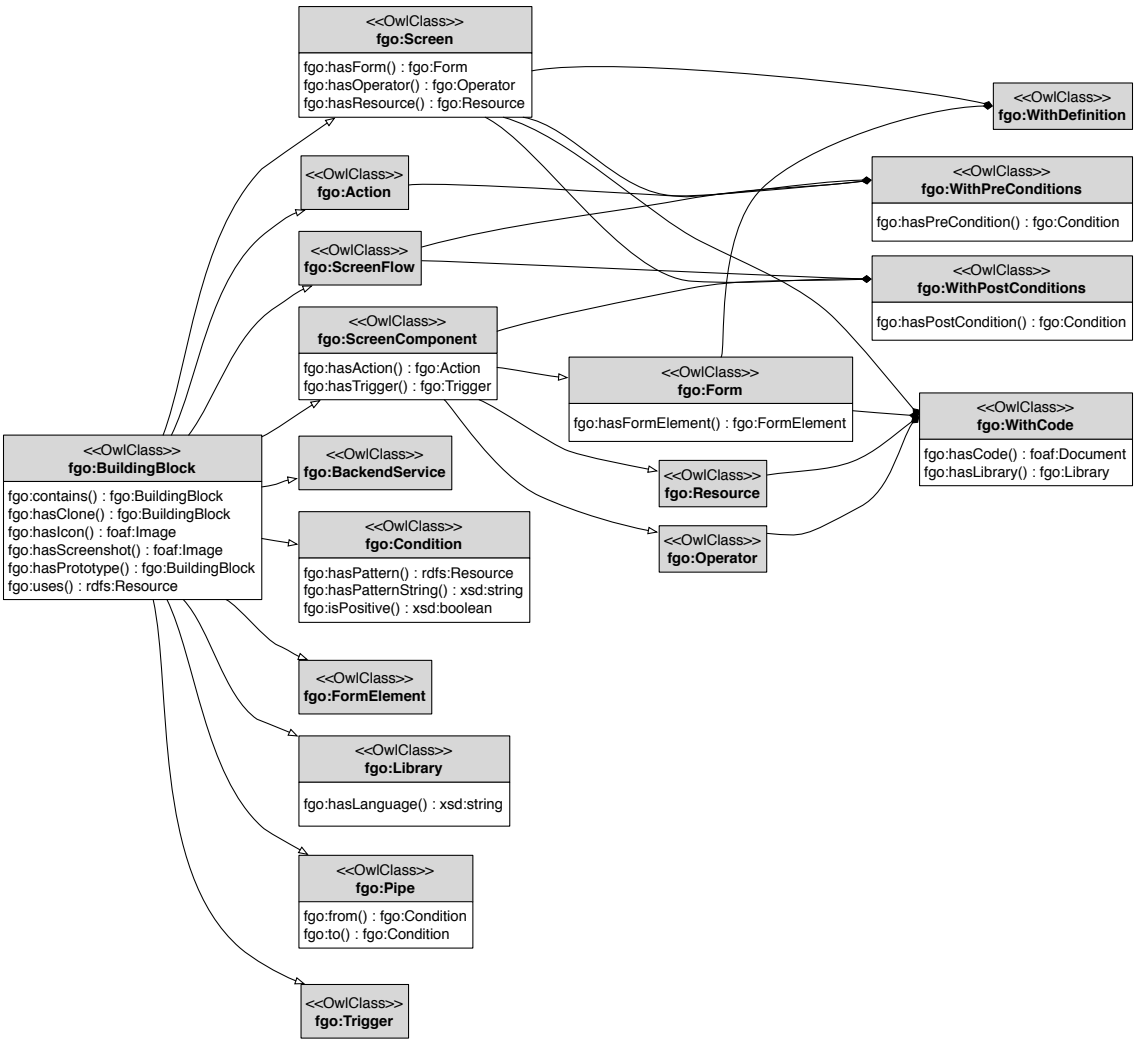


Figure 8: FAST Gadget Ontology Class Hierarchy

These building blocks are not the fundamental classes of FAST, but instead they are specific instances of those classes. In other words, users do not combine the idea of a screen with the idea of a service, but instead they combine the specific "Log In Screen" with the "Amazon Web Service". Technically, these concrete building blocks are implemented as instances of the fundamental classes, specified by asserting particular settings such as label or icons, pre- and post-conditions, etc.

- **Instantiation within a Screenflow at Design-time**: Once a user selects a specific building block in the GVS and adds it to their new screen or screenflow project, they are in principle again instantiating from a conceptual to a concrete level: from the "Log In Screen" as such to a particular log in screen as used in that particular screenflow. Also in this stage, resources are implemented technically as instances.

- **Runtime**: At the final level of instantiation, the screenflow has been packaged as a gadget, deployed to a gadget platform and is now being used by an end-user. At this level, all gadget components exist only in the form of Web standards compliant code (HTML, JavaScript, CSS). For performance reasons, the FAST catalogue transforms any metadata still necessary, as well as definitions for screen pre- and post-conditions, from their original RDF representations into the JavaScript Object Notation (JSON)[4]. The JSON structures used at this level are described in [Palaghita and Rivera, 2010]. At this level, the resources that make up the gadget have moved out of the scope of the FAST platform and therefore out of the scope of this deliverable.

When implementing these four stages, as illustrated in Fig. 9, in OWL DL semantics, we are faced with an immediate problem. Conceptually, we would like to say that $Screen$ is a class, i.e., the set of all screens. As argued above, $LogInScreen$ would then be an instance of $Screen$ ($LogInScreen \subset Screen$), while a particular $LogInScreen_x$ in a particular screenflow would again be an instance of $LogInScreen$ ($LogInScreen_x \subset LogInScreen$). However, unless we want to move into OWL Full semantics, classes cannot be used as instances at the same time, so we cannot use this modelling approach. Furthermore, when we instantiate a resource such as a screen within a particular screenflow, we want to imply that the new instance ($LogInScreen_x$) has the same features such as label, icon or conditions, as $LogInScreen$. In other words, what we really require is a something like a *copy* or a *clone* of the original $LogInScreen$.

[4] http://www.json.org/

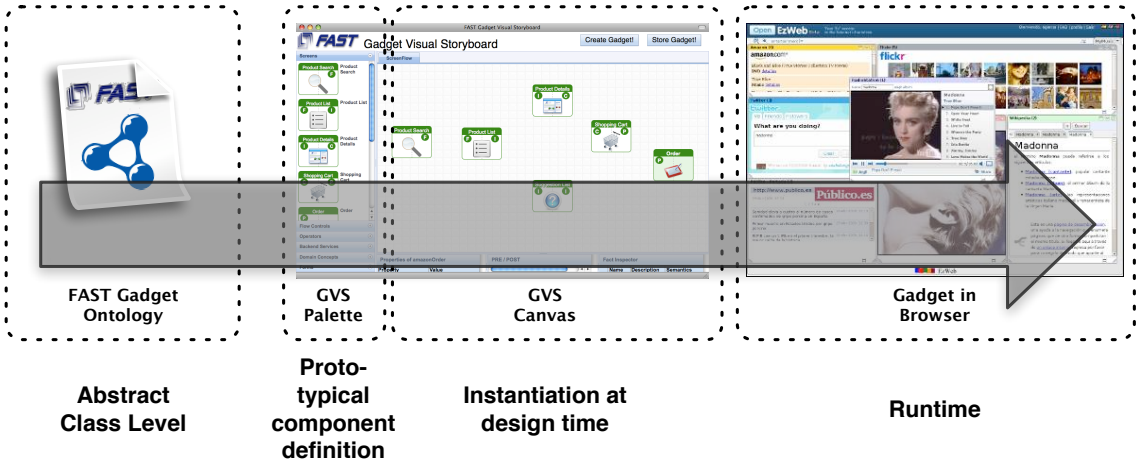


Figure 9: Different levels of representation of FAST building blocks

4.2.1 Prototypes

Neither OWL nor RDFS semantics offer functionality to represent a relationship such as the clone of a resource formally. However, we can find inspiration for this kind of semantics in other areas, such as programming languages, and more particularly in object-oriented programming languages (OO). In OO, there is a general dichotomy between *class-based* languages and *prototype-based* languages. Java, C++ and Smalltalk are examples of the former, while JavaScript[5] (as the most prominent) and Self (as the oldest) are examples of the latter. Both paradigms address the issue of defining similarity between objects, but do so in a different way. The class-based paradigm requires an a-priori classification and categorisation of the world (the class hierarchy), into which individual objects (the instances) are then grouped. All instances of a class inherit its attributes and functionality. Changing this behaviour usually involves changing the class. Prototype-based languages, on the other hand, do not assume prior categorisation. These languages do not show a class vs. instance distinction. Instead, all objects have the same status, and new objects can be *cloned* from other objects, which function as *prototypical objects* in this situation, thereby starting with a specific example and generalising from there. The new object (or clone) will inherit all attributes and functionality from the prototype. Changing behaviour can be done on each object individually. As an aside, the difference between the two paradigms can be traced all the way back to classical philosophy. In this sense, the class-based paradigm has its roots in Plato (ideal forms) and Aristotle (classification of things), while the origin of the prototype-based

[5]The name of the standard is ECMAScript, but we will use JavaScript in this deliverable, since it is the most prominent dialect.

```
// the prototype login screen:
var loginScreen = {label: "Login Screen", icon: "login.png"};

// the clone login screen:
var loginScreen_74382 = Object.create(loginScreen);
loginScreen_74382.label;    // resolves to "Login Screen"

// overwriting the prototype's properties:
loginScreen_74382.label = "Amazon Login Screen";
loginScreen_74382.label;    // resolves to "Amazon Login Screen"

// Note: The create() function is not part of standard JavaScript,
// but a widely used short-hand for the rather more convoluted
// method of the original mechanism.
```

Listing 2: Creation of a new object from a prototype in JavaScript

paradigm only appeared much later as a criticism of the classical model, e.g. by Wittgenstein (the idea of *family resemblance*) and Eleanor Rosch, who introduced prototype theory within cognitive science. While most language designers, let alone programmers, are probably unaware of this heritage [Taivalsaari, 1997], it underlines the importance of this distinction to see how far back it goes.

4.2.2 Prototype-based Semantics for RDF

While of course RDFS and OWL semantics are not identical to class-based OO semantics (the open and closed world assumption being one case in point), they are quite similar in their distinction between classes and instances, and the a-priori categorisation of the world. Along these lines, what we need for FAST is a similar RDF equivalent for prototype-based OO semantics. To achieve this, let's first look again at what creating a new object from a prototype means in a prototypical programming language: essentially, this process means to *clone the prototype*, i.e., to copy all its properties and attributes to the new object[6]. Following our login screen example, a snippet of JavaScript code of what this can look like in practice is shown in List. 2. The `loginScreen` object is created with a number of properties, such as `label` and `icon`. A second `loginScreen_74382` object is then created using the `create` function, using `loginScreen` as its prototype. `loginScreen_74382` inherits all properties from its prototype, but can also overwrite or add individual ones.

[6]Whether or not this is true copy or merely a pointer is implementation-dependant. However, this is usually transparent for the user.

Determining the Prototype's Sub-graph In an attempt to define prototype-based RDF semantics, we first have to specify what the RDF analogue to an object from OO is. Tentatively, we could say that the resource URI is analogous to the object. However, this will not help us when we want to create a new resource C_{URI} from a prototypical resource P_{URI}: a URI merely *identifies* a particular resource, without defining which statements belong to it. We will not know which properties should be cloned, or rather, which new RDF statements should hold for the clone. An RDF graph is simply a set of RDF triples, without any internal boundaries reflecting which of these triples make up a given real-world object (such as a login screen). Exactly which triples these are is difficult to determine. A number of approximations are possible. For the remainder of this discussion, let o be a real-world object, URI_o the URI that identifies it, and G_o the RDF graph containing all triples that are considered part of o. Also, let s be an arbitrary RDF statement, and let $Subj(s)$, $Pred(s)$ and $Obj(s)$ be the subject, predicate and object of s, respectively. The simplest approximation of which triples comprise a real-world object o would be to select all those triples in which URI_o is either subject or object[7]:

$$G_o := \{\ s \mid URI_o = Subj(s) \text{ or } URI_o = Obj(s)\} \tag{1}$$

However, this definition can easily (i) contain false positives (triples that point to URI_o, but are not considered part of o), (ii) lack important triples (e.g., in the case where the object in a triple $< URI, Pred, Obj >$ is itself a complex object), or (iii) have dangling blank nodes. At least the dangling blank node problem can be addressed by extending the Def. 1 to the *minimal self-contained graph* (MSG) [Tummarello et al., 2007], which basically selects the sub-graph so that there are no blank node as leafs. Unfortunately, the MSG still does not necessarily give us the desired sub-graph, since problems (i) and (ii) are not addressed. At the end of the day, these issues are highly context- and interpretation-dependant, which is why there cannot be a deterministic way to know which triples in a graph belong to a particular real-world object. In the general case, we will therefore simply leave the definition of G_o undefined, to be solved individually for each application and use case.

In many cases, a SPARQL graph pattern would be well suited for defining G_o. Depending on the form of the query and the optimisations available on the query processor, evaluating this graph pattern directly might not scale, so that a programmatic solution should be preferred. However,

[7] Note: there is potential for confusion between the term *real-world object* (a tree, a person, a login screen) and the *object* of an RDF triple. The two are two entirely different things!

```
@prefix fgo: <http://purl.oclc.org/fast/ontology/gadget#> .

{
  # the screen is identified by URI_p
  # select all triples with URI_p in subject position
  # this will match basic annotations, such as label, icon, etc.
  { ?URI_p ?p ?o . }
  UNION

  # select all conditions of the screen (both pre- and post), as
  # well as all triples with ?cond in subject position.
  {
    ?URI_p ?p2 ?cond .
    ?cond a fgo:Condition ;
       ?p_cond ?o_cond .
  }
  UNION

  # select the pipes and triggers of the screen.
  {
    ?URI_p fgo:contains ?x .
    ?x fgo:from ?source ;
       fgo:to ?target.
  }
}
```

Listing 3: SPARQL graph pattern to select the object graph of a screen in FAST

even in those cases defining G_o with a SPARQL graph pattern is still useful for its explanatory benefits. For the FAST use case of defining the object graph of a prototypical building block p, the definition of G_p is as given in List. 3 (here showing the definition for a screen; other types of building blocks are slightly different). In essence, the object graph we would like to define for a screen contains its basic annotations (label, icon, etc.), its pre- and post-conditions and its pipes and triggers.

Deriving the Clone Graph Now that we know the (application-dependant) graph G_p of the prototypical object p, we can define what it means to create a new graph G_c for its clone c. In prose, the clone graph will contain all statements of the prototypical graph, but all mentions of URI_p will be replaced with URI_c. Formally, we say that this mapping of G_p into G_c is the prototype mapping pm, defined as follows:

$$pm : G_p \mapsto G_c := \begin{cases} < URI_c, Pred, Obj >, \text{ if } URI_p = Obj(s) \\ < Subj, URI_c, Obj >, \text{ if } URI_p = Pred(s) \\ < Subj, Pred, URI_c >, \text{ if } URI_p = Subj(s) \\ < Subj, Pred, Obj >, \text{ if } URI_p \notin s \end{cases} \quad (2)$$

```
@prefix fgo: <http://purl.oclc.org/fast/ontology/gadget#> .

SELECT DISTINCT ?sub_res
WHERE {
  {
    ?URI_p ?p ?sub_res .
    ?sub_res a fgo:Condition
  }
  UNION
  {
    ?URI_p fgo:contains ?sub_res .
    ?sub_res fgo:from ?source ;
       fgo:to ?target.
  }
}
```

Listing 4: SPARQL SELECT to select sub-resources of a screen in FAST

However, there are cases in which this mapping will not be sufficient. More specifically, when the prototype's object graph contains resources which themselves need to be cloned, their URIs need to be replaced in the same fashion as shown in Def. 2. An example of this in FAST is the cloning of screens: the URIs of conditions, triggers and pipes are based on their screen, and therefore need to change along with it. As discussed for the definition of the object graph above, selecting which sub-resources need to be cloned is application-dependant. While we do not propose any particular mechanism for selecting sub-resources, also here a SPARQL graph pattern is a possibility, as shown in List. 4. Here, the replacement algorithm would have to be performed for each `?sub_res`.

Finally, coming back to the levels of representation in FAST, List. 5 shows the class level, prototypical instance and cloned instance of a screen, using TriG syntax [Bizer and Cyganiak, 2004] (an extension of Turtle syntax for named graphs). Using the terminology established so far, URI_p is `catalogue:LogInScreen` and URI_c is `catalogue:LogInScreen_74382`, while the corresponding object graphs G_p and G_c are `catalogue:prototype_graph` and `catalogue:clone_graph`, respectively. Prototype and clone refer to each other using the `fgo:hasPrototype` and `fgo:hasClone` properties.

Figure 10 illustrates the same example graphically, using different colours to represent the two different object graphs: the red graph G_p in the background represents the graph of the prototypical instance, whereas the green graph G_c in the front in represents the new clone as it would be used in a concrete screen flow.

```
@prefix fgo: <http://purl.oclc.org/fast/ontology/gadget#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix catalogue: <http://fast.morfeo-project.org/catalogue/> .
@prefix dcterms: <http://purl.org/dc/terms/> .

# the class of screens, as defined in the FAST gadget ontology
fgo:Screen a owl:Class .

# an instance of Screen, the log in screen.
# this acts as a prototype for screens used in screenflows.
# represented in the palette of the GVS
catalogue:prototype_graph {
  catalogue:LogInScreen a fgo:Screen ;
  fgo:hasClone catalogue:LogInScreen_74382 ;
  dcterms:title "Login Screen" ;
  fgo:hasIcon catalogue:login.png .
}

# a clone of LogInScreen as used in a particular screenflow:
# represented on the canvas of the GVS
catalogue:clone_graph {
  catalogue:LogInScreen_74382 a fgo:Screen ;
  fgo:hasPrototype catalogue:LogInScreen ;
  dcterms:title "Login Screen" ;
  fgo:hasIcon catalogue:login.png .
}

#todo - conditions, triggers and pipes are missing
```

Listing 5: Prototyping example — different levels of representation of a FAST screen (code)

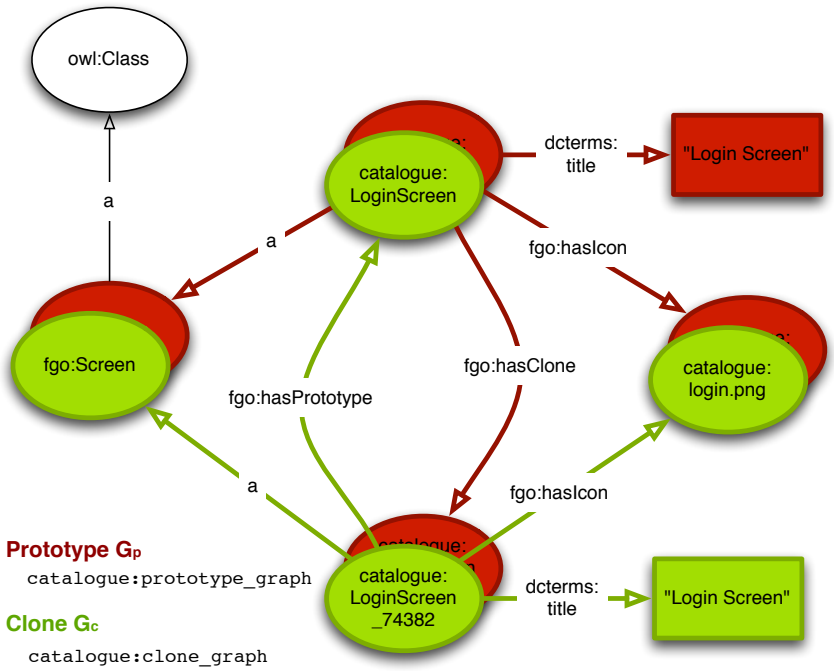


Figure 10: Prototyping example — different levels of representation of a FAST screen (diagram)

4.2.3 Discussion

This section proposed an approach for prototype-based semantics for RDF. This was done against the backdrop of different levels of building block representations within FAST, but could have much wider use. Business function modelling and other use cases have been mentioned in private discussions. E.g., there might be a class `BusinessFunction` with a concrete instance `ShippingBF`. If we want to say that there are further specialisations of `ShippingBF`, such as a `FedexBF` and a `TNTBF`, the usual class-based modelling paradigm won't help us, but prototype-based modelling will [Bhiri, 2010].

Additionally, it should be pointed out that in practice FAST does not currently make use of the prototyping mechanism to the fullest extent. Certain situations *require* the creation of new objects from prototypes (e.g., the use of several operators of the same kind in a single screen during the screen design phase), while other situations simply benefit from the approach because it leads to a more coherent URI minting scheme (e.g., the cloning of conditions, pipes and triggers for screens). However, in yet other situations no new objects are generated, since there is no compelling reason to do so (e.g., screen components such as forms, operators and service resources are not considered part of a screen's object graph, and are therefore not cloned with it). With this in mind, FAST should be considered a first "testing in the field" of our proposal for prototype-based semantics, rather than a fully-fledged implementation.

Furthermore, it should be mentioned that our proposal makes no assumption on how the cloning process is actually implemented, i.e., whether any cloned triples are actually materialised into the database, or if the cloning process should be handled as inference rules on top of the RDF layer. In the FAST catalogue, we are currently following the former approach.

Finally, in previous versions of this deliverable we used the terms *template* and *copy* instead of *prototype* and *clone*. The change to the new terminology was done because it is much more established and well-defined. Also, the term *RDF template* had been used in prior work in the community, but with a different meaning. [Davis, 2003] use the term in analogy to XSLT, but acting on an RDF graph rather than an XML tree. [Kawamoto et al., 2006] present a semantic wiki, where templates help non-technical users to create semantic descriptions.

```
@prefix fgo: <http://purl.oclc.org/fast/ontology/gadget#> .
@prefix catalogue: <http://fast.morfeo-project.org/catalogue/> .
@prefix dcterms: <http://purl.org/dc/terms/> .
@prefix doap: <http://usefulinc.com/ns/doap#> .

# this is a particular type of screen:
catalogue:LogInScreen a fgo:Screen ;
  # the dcterms vocabulary should be used for a a number of basic annotations.
  # literals should if applicable be typed to a particular language, thus
  # making multiple localisations possible.
  dcterms:title "Log In Screen"@en-gb ;
  dcterms:description "This screen lets the user log in to the system."@en-gb ;

  # time literals should be formatted according to ISO8601 and typed accordingly:
  dcterms:created "2009-11-23T13:39"^^xsd:dateTime ;

  # for graphical representations of the resource within the GVS we use
  # sub-properties of foaf:depiction in the FAST gadget ontology namespace:
  fgo:hasIcon catalogue:login.png ;
  fgo:hasScreenshot catalogue:login-screenshot.png ;

  # versioning information can be handled with a simple literal:
  doap:revision "0.75b" .
```

Listing 6: Example of basic annotation of a FAST resource

4.3 Basic Annotation

In this section, we will present the use of some basic resource annotation properties in FAST. In general, since screenflows, screens and their components are sub-classes of `fgo:Resource`, all properties defined to have a domain of `fgo:Resource` should be used. In addition, a number of terms from external ontologies should also be used to annotate resources. Note that all of these annotations are valid both for the prototypical component definitions (prototypes) and the design time instances (clones). A detailed description of all terms from the FAST gadget ontology namespace can be found in Sect. B of this document.

4.4 FAST Users

In modelling resources in FAST, the kind of users that are relevant are the users of the FAST tools themselves (i.e., the GVS), as opposed to end users of the gadgets. The following List. 7 shows how an individual user is modelled, using terminology from the FOAF and SIOC vocabularies.

Once a user has been defined in this way, they can for example be used to specify the creators of

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix sioc: <http://rdfs.org/sioc/ns#> .
@prefix people: <http://fast.morfeo-project.org/catalogue/person/> .
@prefix users: <http://fast.morfeo-project.org/catalogue/user/> .

# the user as such, independently from its role in FAST, is described
# as an instance of foaf:Person. Any kind of further description is
# possible, but only certain basic facts will be picked up by the GVS.
people:ismael a foaf:Person ;
  foaf:name "Ismael Rivera" ;
  foaf:depiction catalogue:ismael_01.jpg .

# the user's role in FAST is described as an instance of sioc:User
users:ismael a sioc:User ;
  foaf:account_name "FAST GVS" ;
  foaf:accountServiceHomepage <http://fast.morfeo-project.org/gvs> .

# the person and user instances are linked as follows:
people:ismael foaf:holdsAccount users:ismael .
users:ismael sioc:account_of people:ismael .
```

Listing 7: Basic description of a user in FAST

```
@prefix fgo: <http://purl.oclc.org/fast/ontology/gadget#> .
@prefix catalogue: <http://fast.morfeo-project.org/catalogue/> .
@prefix dcterms: <http://purl.org/dc/terms/> .

catalogue:LogInScreen a fgo:Screen ;
  dcterms:creator users:ismael .
```

Listing 8: Modelling the creator of a resource

individual screen and screenflow resources, as shown in List. 8. This kind of creation metadata is being added to other metadata as shown in Sect. 4.3.

4.5 Defining Pre- and Post-conditions

The facts which define the pre- and post-conditions of screens and screenflows in FAST are modelled as individual RDF triples. In effect, this means that conditions, which are sets of facts, are modelled as graph patterns using SPARQL notation [Prud'hommeaux and Seaborne, 2008]. E.g., a simple pre-condition such as *"There has to be a user"* could be expressed as in List. 9. Literally, this very simple pattern means *"a variable ?user is of type sioc:User"*. In logical terms, it means something along the lines of *"there exists a sioc:User"*. In order to determine if this pre-condition can be fulfilled, it needs to be executed against the RDF graph in question. This graph could for example consist of the set of post-conditions of all screens currently available on the

```
?user a sioc:User .
```

Listing 9: Simple pre-condition

```
?user a sioc:User ;
      foaf:accountName ?account_name .
?person a foaf:Person ;
      foaf:holdsAccount ?user ;
      foaf:name ?person_name .
```

Listing 10: Post-condition at the prototypical stage

canvas.

Post-conditions are expressed in a similar fashion. As part of prototypical instances, post-condition patterns have variables in the same way as pre-condition patterns. When a building block is cloned to the canvas, the pattern needs to be materialised. To do this, each variable will be replaced with a randomly generated URI or blank node. At runtime, variables are then replaced with actual values once the screen for which the post-condition holds has been executed. As an example, consider the post-condition of a login screen. In natural language, it could be *"Once the login process has finished, there will be a user object"*. Using the same notation as before, this could be expressed as shown in List. 10, extended with some additional facts (*"There is a user object which has an account name. There is also a person which has a name, and which has the user object as an online account."*).

Once added to the canvas in the GVS ("instantiated at design time"), each variable is then replaced with a blank node identifier, as shown in example 11. This is necessary because, while variables are available in SPARQL, they cannot be represented directly in RDF. The pre-condition in example 9 could now be matched successfully as a SPARQL query against the canvas graph.

Finally, during runtime, the post-condition's variables would be replaced with the actual values of the user which completed the login screen ("instantiated during runtime"). E.g., this could result

```
_:b0 a sioc:User ;
      foaf:accountName _:b1 .
_:b2 a foaf:Person ;
      foaf:holdsAccount _:b0 ;
      foaf:name _:b3 .
```

Listing 11: Post-condition as instantiated during design time (on the FAST GVS canvas)

```
        <http://fast.org/gvs/knud> a sioc:User ;
              foaf:accountName "dunk" .
        <http://kantenwerk.org/metadata/foaf.rdf#me> a foaf:Person ;
              foaf:holdsAccount <http://fast.org/gvs/knud> ;
              foaf:name "Knud Moeller" .
```

Listing 12: Post-condition during runtime

```
@prefix fgo: <http://purl.oclc.org/fast/ontology/gadget#> .
@prefix catalogue: <http://fast.morfeo-project.org/catalogue/> .
@prefix dcterms: <http://purl.org/dc/terms/> .
@prefix sioc: <http://rdfs.org/sioc/ns#> .

# linking a screen to a post-condition
catalogue:LogInScreen a fgo:Screen ;
  dcterms:title "Log In Screen"@en-gb ;
  fgo:hasPostCondition catalogue:post_condition_32187 .

# defining the facts of the post-condition
catalogue:post_condition_32187 a fgo:Condition ;
  fgo:hasPattern catalogue:pattern_4329082 ;
  fgo:hasPatternString "?user a sioc:User ." ;
  fgo:isPositive true .

# named graph of a particular condition pattern
catalogue:pattern_4329082 {
  _:x74838 a sioc:User .
}
```

Listing 13: Linking screens to conditions

in a graph such as in example 12. However, in the current implementation of FAST, gadgets do not use an RDF representation of facts at runtime, but instead a translation of the facts into the JSON format.

Regarding the representation of condition graph patterns and their linking to building blocks in the FAST backend, two parallel approaches are being followed. Each graph pattern is (i) represented as a named graph which is linked from an `fgo:Condition` using the `fgo:hasPattern` property, and (ii) as a string, linked from the condition using the `fgo:hasPatternString` property. This second approach is followed for convenient rendering in user interfaces and to allow optional support the use of a triple store without support for named graphs. Each such condition will be linked to a screen or screenflow using either the `fgo:hasPreCondition` or `fgo:hasPostCondition` property. A concrete example of this principle is given in List. 13.

Since SPARQL has no direct support for negation, and since there is a need to allow screens to remove facts from the canvas graph as well as adding them, both pre- and post-conditions can be modelled to be either *positive* or *negative* using the `fgo:isPositive` property. For pre-

conditions, the interpretation of *positive* is that the condition must hold, while *negative* means that the negation of the condition must hold. For post-conditions, *positive* leads to the instantiated condition being added to the canvas, while *negative* will remove the instantiated condition. These interpretations are summarised in Tab. 5.

Table 5: Different semantics for `isPositive`

	pre-condition	**post-condition**
isPositive: true	condition must be fulfilled	condition instantiated
isPositive: false	¬condition must be fulfilled	instantiated condition removed

4.6 Conditions and Piping within Screens

In year two of the FAST project, work has been started to extend the flexibility of the GVS to also include the creation of new screens from building blocks, as opposed to only assembling pre-made screens into screen flows. The gadget ontology supports this by modelling screen internals such as *forms*, *service resources* and *operators*. Each of those building blocks can have their own pre- and post-conditions (either directly or through their *actions*). However, while conditions on the screen/screen flow level are matched automatically and the flow of data is establish dynamically, on the screen-internal level this is defined explicitly through *pipes*. Pipes are themselves components of a particular screen and connect matching post- and pre-conditions of other components in the same screen.

The concept of piping on this level of the FAST platform is illustrated graphically in Fig. 11, and in code in List. 14[8]. The screen is composed of (possibly among other things) a backend service resource, which wraps the Amazon search service, as well as a frontend Web form, which shows a list of Amazon products to the user. Additionally, the screen contains an instance of `fgo:Pipe`, which links the resource's post-condition (a product list) to the matching pre-condition of the form's "showTable" action. The desired functionality of this arrangement is that, whenever the service has returned a list of products in response to a search, this list will be piped to the form, where it

[8]The example does not show a complete screen definition, but only the aspects which are relevant for this section.

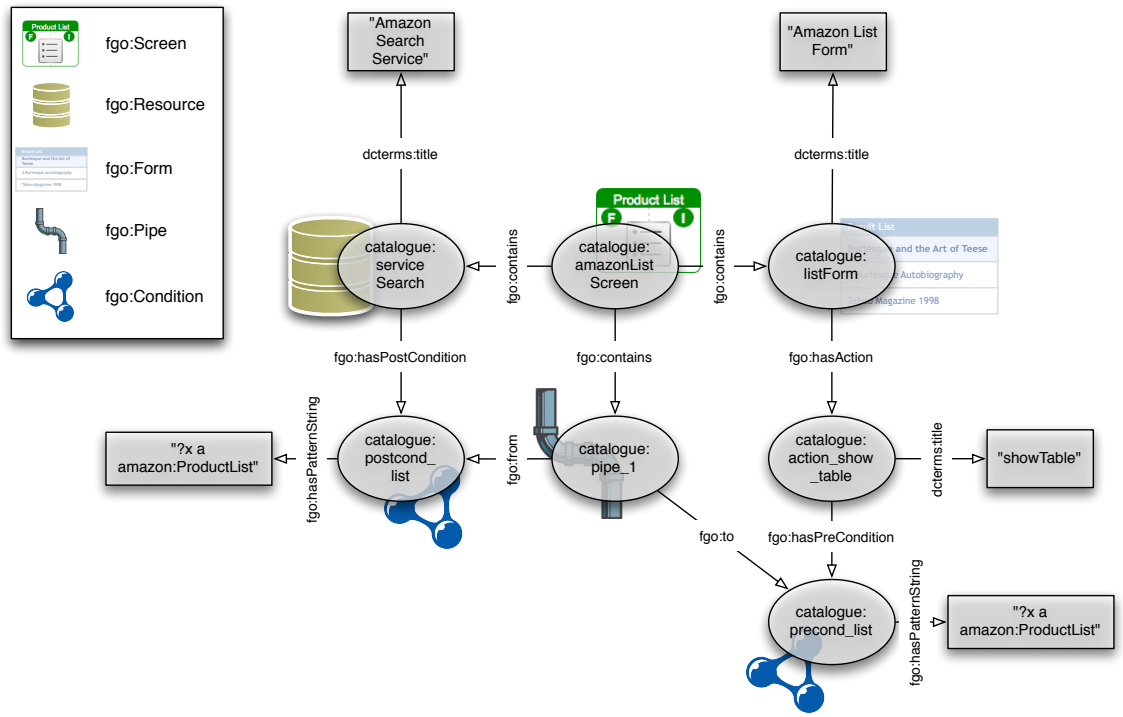


Figure 11: Explicit piping within a screen from a post- to a pre-condition (diagramme)

will be displayed to the gadget's user.

4.7 Extension towards Specific Domains

While the FAST gadget ontology provides the framework and scaffolding for the formal description of complex gadgets, any tailoring of a gadget or building block for a specific domain requires the integration of external domain ontologies. These ontologies can either be existing domain ontologies (e.g., the Good Relations ontology [Hepp, 2008][9] for e-Commerce) or ontologies which were purpose-built for FAST. There are three main entry points at which external ontologies come into play:

- *Domain Contexts:* By providing a domain context for a screen (or any other FAST building block), it is possible to express what domain this resource is relevant for. Examples for domain contexts are "medicine", "EU projects", "catering", "human resources", "books", etc. As

[9] http://www.heppnetz.de/projects/goodrelations/

```
@prefix fgo: <http://purl.oclc.org/fast/ontology/gadget#> .
@prefix catalogue: <http://fast.morfeo-project.org/catalogue/> .
@prefix dcterms: <http://purl.org/dc/terms/> .

# the screen and its components:
catalogue:amazonListScreen a fgo:Screen ;
  dcterms:title "Amazon List Screen"@en ;
  fgo:contains catalogue:serviceSearch , catalogue:listForm , catalogue:pipe_1 .

catalogue:serviceSearch a fgo:Resource ;
  dcterms:title "Amazon Search Service"@en ;
  fgo:hasPostCondition fgo:postcond_list .

catalogue:listForm a fgo:Form ;
  dcterms:title "Amazon List Form"@en ;
  fgo:hasAction catalogue:action_show_table .

# actions and conditions:
catalogue:action_show_table a fgo:Action ;
  dcterms:title "showTable"@en ;
  fgo:hasPreCondition fgo:precond_list .

catalogue:precond_list a fgo:Condition ;
  fgo:hasPatternString "?x a amazon:ProductList" .

catalogue:postcond_list a fgo:Condition ;
  fgo:hasPatternString "?x a amazon:ProductList" .

# an explicit pipe from service to form:
catalogue:pipe_1 a fgo:Pipe ;
  fgo:from catalogue:postcond_list ;
  fgo:to catalogue:precond_list .
```

Listing 14: Explicit piping within a screen from a post- to a pre-condition (code)

```
@prefix fgo: <http://purl.oclc.org/fast/ontology/gadget#> .
@prefix catalogue: <http://fast.morfeo-project.org/catalogue/> .
@prefix ctag: <http://commontag.org/ns#> .
@prefix dcterms: <http://purl.org/dc/terms/> .

# defining the domain context of a screen with common tag:
catalogue:LogInScreen a fgo:Screen ;
        dcterms:title "Amazon Login Screen"@en ;
        ctag:tagged catalogue:tag_42389.

# defining the meaning of a tag through reference to DBpedia:
catalogue:tag_42389 a ctag:Tag ;
        ctag:means <http://dbpedia.org/resource/Amazon.com> ;
        dcterms:title "Amazon.com"@en .
```

Listing 15: Defining a screen's domain context using the Common Tag vocabulary

defined in the Integration Document in Sect. 3.3.5, we integrate the `ctag:tagged` property from the Common Tag vocabulary to define domain contexts as complex tags. As discussed in Sect. 3.3.4, one option for finding domain identifiers to be used with this property is the use of DBpedia identifiers such as `http://dbpedia.org/resource/Medicine` (DBpedia is a large RDF corpus automatically extracted from the info boxes of Wikipedia entries). This is now a widely used and recommended practice in the Semantic Web community. Similar corpora such as FreeBase could also be used. A concrete example of defining the domain context of the login screen to the Amazon service is given in List. 15, showing the use of `ctag:means` to disambiguate the meaning of the tag "Amazon.com".

- *User Interests:* In the same way as domain contexts, a user's interests can be specified using DBpedia identifiers and structured tagging. Matching domain contexts to used interests, the GVS (through the catalogue) can then suggest relevant resources to the user.
- *Pre- and Post-conditions:* In specifying pre- and post-conditions, terminology for an open-ended number of domains is necessary. While a number of classes will be reused quite often, very specific screens will require very specific vocabulary. We cannot, as part of the FAST project, provide vocabulary for all conceivable domains of discourse, and while also here DBpedia identifiers may be an option, often more precise terms will be needed.

It should be added that for any given screenflow, the concrete definition of both domain context could be derived automatically from any semantic description which the backend services might have. While this might happen directly, without any transformation or mapping, this approach would be impractical in practice. The reason for this is that each backend service can potentially (and very probably) express its semantics in different ontologies or vocabularies, which would

render the pre- and post-condition mechanism in FAST useless. Therefore, we assume that a mapping or mediation layer will ensure that the different semantics at the backend service level will be mapped to a common set of terms within the FAST platform. This problem is the topic of deliverable 2.4 in FAST [Ambrus, 2010] and has also been discussed in [Ambrus et al., 2009].

5 Evaluation

The FAST ontology is continuously being evaluated through its direct and indirect employment in the implementation work packages WP3-5, as well as through an ongoing dialogue between the ontology designers and the platform implementers. The use case-based evaluation performed in WP6 provides additional input to this process. This kind of evaluation has lead to the gradual change, extension and adjustment of the documents produced in the different stages in the ontology development process, such as the glossary of terms in the conceptualisation phase, the integration document and the formal OWL ontology document. In addition to this kind of continuous evaluation we have also looked into other kinds of metrics ontology evaluation, as discussed in the following paragraphs.

5.1 Usability as a General Design Principle

Apart from "hard" metrics for the evaluation of ontologies, so-called "soft" factors for ontologies are relevant for facilitating better understanding by users and implementers, and therefore ultimately greater usability and uptake. [Möller et al., 2010] define five such soft factors as guidelines for ontology developers:

- **reuse external ontologies** to facilitate integration and compatibility,
- **avoid over-engineering** and keep the number of ontology terms down to reduce the cognitive load on implementers,
- **be a good Web citizen** by following the principles of linked data,
- **document well** to make it easy for implementers to understand and use the ontology correctly and
- **provide tool support** to ease the work-load involved in generating instance data.

In the following, we will look at each of those guidelines and discuss how far the FAST gadget ontology fulfils them.

5.1.1 Reuse of Existing Ontologies

An important aspect to consider when designing any ontology which is aimed to be used not only within a restricted institution or community is to reuse terms from existing ontologies and vocabularies as much as possible. Reuse serves the purpose of lowering the entry barrier for third parties to adopt the ontology — if they can use familiar concepts and terms, they will be more inclined to try out something new. Equally important is the fact that reuse of terms facilitates data integration, which, after all, is one of the major goals of the Semantic Web at large.

Rather than reinventing the wheel in the various terms that have been defined in its requirement specification, the FAST gadget ontology integrates a number of well-established external ontologies. In particular, FOAF and SIOC are used to model tool users and developers, Dublin Core is applied for its general purpose annotation vocabulary and Common Tag is integrated to allow the modelling of a FAST resource's domain context as complex conceptual tags. Details are given in Sect. 3.3.

5.1.2 Avoid Over-engineering

The concrete interpretation of this criterion depends to a large degree on the intended use case and user community of the ontology. For a large-scale undertaking spanning many domains, and where reasoning is in the centre of attention — CyC[10] e.g. represents this type of ontology — a certain amount of complexity is necessary. However, in order to appeal to adopters who are familiar with formal knowledge representation, simplicity can be an important factor. Good examples of such ontologies are FOAF and SIOC, which would arguably not have had such an impact in the wider Web community if they tried to model their respective domains in a much more complex way, making use of the full range of expressivity offered by e.g. the RDFS and OWL ontology languages. Indirect support for this argument is provided by [Wang et al., 2006], who show that most ontologies found on the Web are on the lower end of the expressivity spectrum, indicating that practitioners either don't need or don't understand the full range of possible expressiveness.

At the moment, the use case for the FAST gadget ontology is the tool-internal representation of gadgets and their components, which means that simplicity for users in the wider community is

[10] http://cyc.com/cyc/opencyc

only of secondary importance. However, the ontology is still kept relatively basic, both in terms of the number of terms (currently 18 classes and 22 properties), and in the complexity of the class definitions themselves, which are mostly restricted to a straightforward class subsumption hierarchy, with the exception of the sparse use of the boolean class expression `owl:unionOf`.

As another measure to avoid over-engineering, it was ensured that the logical complexity of the ontology falls within the DL fragment[11] of OWL. While this does not necessarily ensure that an ontology is easy to comprehend by users, it can help to achieve this goal.

5.1.3 Be a Good Web Citizen

While ontologies have gained much attention as a key component in the emerging Semantic Web, not all ontologies are meant for use on the Web. However, if ontologies are supposed to be deployed on the Web and used in a Web context, they should also be a good Web citizen, meaning they should follow certain rules and best practices, in particular those defined for the Web of linked data [Berners-Lee, 2006]. Of those rules, the first two — all things are identified with a URI and URIs should be HTTP URIs — are followed by definition if the ontology in question is formalised in RDF. The third one, however — that information should be served on the Web for each URI — has often been neglected in the past. Terms in ontologies were identified with URIs which were not dereferencable, i.e., the ontology was defined in a namespace URI, but the ontology document was served at an arbitrary other URI (or even not served anywhere at all). This has two negative effects on adoption: (i) automatic agents encountering instance data for an ontology cannot request the ontology from its namespace URI to learn the precise formal definitions of the terms used, and (ii) human users encountering terms from an ontology cannot look up definitions and documentation for them in a Web browser.

The FAST gadget ontology fulfils the previous criterion by following the best practices recommendations outlined in [Berrueta and Phipps, 2008], thereby ensuring that the ontology is available within a stable namespace and URI and can be accessed both by machine agents to retrieve its formal definition, as well as by human agents, who will retrieve a human-understandable HTML documentation from the same URI. Furthermore, the fourth rule of linked data (links should be established between datasets) is covered through the integration of external ontologies, as dis-

[11] We use the Pellet reasoner: `http://www.mindswap.org/2003/pellet/demo` to measure complexity.

cussed above in Sec. 5.1.1.

5.1.4 Document Well

A lack of good documentation for an ontology will make it hard for potential adopters to comprehend and make use of it, whereas on the other hand, good documentation can be the deciding factor in which ontology gains more uptake. On the most basic level, documentation can be provided by the use of annotation properties within the formal definition of the ontology itself (e.g., `rdfs:comment`). However, in practice this should be extended with human-readable documentation, ideally at the available online at the namespace URI of the ontology itself (see above in Sect. 5.1.3). The human-readable documentation can be as simple as a list of terms and explanations generated from the ontology source, e.g., with a tool such as *VocDoc*[12], which was developed specifically to generate documentation for the FAST gadget ontology both in an online format and a print format. Much more useful than a simple list of terms, however, are concrete instance examples which illustrate the actual use of the various ontology terms in practice.

Our ontology follows the documentation guideline, both through its online documentation, as well as through this deliverable. The two versions of the documentation provide both a lookup list for terms and their definitions, as well as concrete examples of term usage in instances.

5.1.5 Provide Tool Support

A further guideline to ease and encourage adoption of an ontology is the provision of tool support to users, which will help them generate instance data. Examples of such tools can be online forms such as FOAF-a-matic[13] for FOAF or plugins for popular content management systems to generate SIOC data[14]. In FAST, instance data for the FAST gadget ontology is never intended to be generated by hand, but rather through the use of the FAST catalogue. Additional tool support is therefore not currently necessary.

[12] `http://kantenwerk.org/vocdoc/`
[13] `http://www.ldodds.com/foaf/foaf-a-matic`
[14] `http://sioc-project.org/exporters`

5.1.6 Modularisation

Apart from the guidelines proposed by [Möller et al., 2010], another basic design principle which helps to improve accessibility and understanding of the ontology is modularisation. While all terms reside in the same namespace, classes and properties are nevertheless be grouped according to the specific sub-tasks which they belong to. E.g., terms are grouped into defining *resources*, *pre- and post-conditions*, *general annotation* or *users and user profiles*.

6 Conclusions and Outlook

The current version of deliverable 2.2 provides the foundation for enabling the semantic definition of gadgets within the context of the FAST platform. We have specified what the individual components of a gadget are, and how they interrelate. In developing our ontology, we have followed Methontology, a well-known methodology for ontology development, which we have extended with additional UML diagrams to define the conceptual model of the FAST platform. During the course of this process, a number of living documents — the requirements specification document, the glossary of terms and conceptual model, the integration document and finally the implementation document — were produced. Looking at the integration document, it becomes apparent that we integrate terms for those parts of the conceptual model which are not unique for FAST, i.e., users and user profiles, basic annotation needs or the tagging of building blocks. Aspects of the model which are unique to FAST, however, had been modelled from scratch.

The various documents eventually leading to the FAST gadget ontology serve as tools for development and as documentation at the same time, and are being extended and updated throughout the duration of the project and beyond. The driving force behind these changes is the dialogue between ontology and application developers, as well as the testing and evaluating of the ontology within the FAST project.

Apart from the conceptual model and ontology as such, we have introduced the notion of RDF graph templates as a means to overcome the strict separation of classes and instances in OWL. Within FAST, we use these templates to distinguish between prototypical building blocks and instances of such prototypes within a concrete screen flow. Also, a solution for representing pre- and post-conditions of building blocks such as screens flows, screens and screen components — a central contribution of FAST technology — has been presented, implemented and tested both within the FAST catalogue and the FAST GVS.

The FAST gadget ontology has initially been developed using the original OWL ontology language (or OWL 1). Since the start of the project, OWL 2[15] has been released and is now a W3C recommendation. In order to avoid any potential migration issues, we have so far refrained from adopting OWL 2, but may do so in the future.

The latest version of the ontology and its documentation can always be accessed at `http://purl.oclc.org/fast/ontology/gadget`.

[15] `http://www.w3.org/TR/owl-overview/`

A Development Methodology

A.1 *Methontology*: A Methodology for Ontology Development

In order to put the design of the ontology on a stable footing, we chose to adopt a tried and tested design methodology, rather than using a purely ad-hoc approach. We decided to use the *Methontology* methodology, which was first introduced in [Gómez-Pérez et al., 1996] and [Fernández et al., 1997]. Methontology provides a good balance of formalisation and streamlining of the design process on the one hand, while on the other hand its requirements on how strictly one needs to follow the individual phases and steps are intentionally left relatively loose. The authors specifically advocate an "evolving prototype life cycle", which allows the ontology to "grow depending on its needs" [Fernández et al., 1997].

Methontology consists of seven different stages, which will be briefly described in the following sections. For a more detailed discussion, we point the reader to the original papers proposing this methodology. The phases are *specification*, *knowledge acquisition*, *conceptualisation*, *integration*, *implementation*, *evaluation* and *documentation*. It is important to note that the phases do not have to be visited slavishly in the order presented here. Instead, a much more likely scenario is that the ontology designers will visit each phase roughly in this order, but are free to revisit each phase at any time to apply changes.

A.1.1 Specification

The specification phase sets the stage for the rest of the ontology development process. The central activity in this phase is the setting up of an *ontology requirement specification document*, which will act as a guideline for most of the other phases. The level of formality of the specification document is left open and can range from pure natural language, over a set of intermediate representations to using competency questions. Most likely, it will be a mix of those formats. Regardless of the chosen format of the specification document, it must answer questions regarding the *purpose* of the ontology (intended use and users, scenarios, etc.), its *scope* (what are the things that need to be covered), its *level of formality* (e.g. highly informal, semi-formal or rigorously

formal [Uschold and Gruninger, 1996]) and the *sources of knowledge* used when researching the ontology. It should be noted that the scope as defined in the specification document does not need to be complete. Rather, it should provide a good impression of the kind of terms that need to be covered by the ontology.

The *Domain Analysis* section of this deliverable implements an ontology requirement specification document for the FAST gadget ontology (which is not to be confused with the FAST requirements deliverable [Solero et al., 2010]!).

A.1.2 Knowledge Acquisition

Knowledge Acquisition in the context of Methontology is a loose collective term for all activities which are aimed at gathering background knowledge to clearly determine purpose and scope of the ontology. As such, it feeds directly into the specification phase, but can just as well be relevant at later stages of the design and development process. Similarly to evaluation and documentation, knowledge acquisition is a supporting activity, which takes place throughout the whole development process.

Typical ways of knowledge acquisition are referring to handbooks, encyclopaedias or other written material (through formal or informal text analysis), performing interviews with domain experts (structured or non-structured) or evaluating previous conceptualisation work done for the domain in question.

The primary sources used for knowledge acquisition for the the FAST gadget ontology are referred to in Sect. 2.

A.1.3 Conceptualisation

The conceptualisation phase stands at the core of the ontology development process. The ontology is still in a language-/format-independent state, but is becoming increasingly formal. Based on the specification document, the ontology designers now create a number of increasingly detailed and fine-grained tables and dictionaries which cover all terms to be included in the ontology

(at the current stage - an ontology in the Semantic Web sense is never complete). In the first iteration, a so-called *glossary of terms* (GT) is built, which includes all concepts (or classes), instances and properties (or verbs). The GT does not have to be built from scratch, but can instead be understood as an extension of the scope definition in the specification phase. In contrast to the specification document, the GT is a living document that should always reflect the latest and most complete list of terms. Once completed (in a first iteration), the terms in the GT will be grouped according to relatedness. For each group, the concepts are arranged in a classification tree. Instances, constants and attributes are grouped in corresponding tables, while properties are captured in verb diagrams. If necessary, tables with formulas and rules are set up at the end of the conceptualisation phase.

A.1.4 Integration

The integration phase offers an opportunity for the ontology designer to ensure that their ontology is well integrated with other, existing ontologies. This can either mean connecting the emerging ontology to upper or meta ontologies (in the case that there are appropriate super classes or properties in those ontologies which can act as connection joints), or the reuse of terms from other ontologies (in the case that other ontologies have already defined classes or properties which fall in the scope of the emerging ontology). A typical example for a meta-ontology is OpenCyc, while the FOAF vocabulary [Brickley and Miller, 2007] is an example of an often-reused ontology to describe people. As the output of this phase, an *integration document* is suggested, which gives detailed information about which terms were taken from which external ontology.

A.1.5 Implementation

The implementation phase is what is often (wrongly so) considered to be the main activity in developing an ontology: materialising the various ontology terms in a concrete language, which can be RDFS or OWL for the Semantic Web, or any other formal language in other contexts (even a programming language such as C++). The authors of Methontology do not go into a lot of details regarding the carrying out of this phase, other than that the outcome will be the formalisation of

the ontology in the ontology language of choice (the *implementation document*).

A.1.6 Evaluation

According to [Fernández et al., 1997], evaluation "means to carry out a technical judgment of the ontologies, their software environment and documentation with respect to a frame of reference" (in this case the requirement specification document). Evaluation can take place at any time during the ontology development process, and comprises activities such as *validation* and *verification*.

For the FAST Gadget Ontology, most of the evaluation process will take place as part of the general implementation efforts in WPs 3–5, which will either directly or indirectly make use of the ontology and therefore function as means to test its feasibility, as well as providing new input to its evolution. Additionally, the work carried out in WP6 (Experimentation and Evaluation) in years two and three of the project is providing valuable input to the evaluation of the ontology.

A.1.7 Documentation

In Methontology, documentation is an activity which is automatically carried out throughout the development process, in the form of the various documents which form the output of the individual phases. Since this deliverable contains the current versions of all those documents, it represents a snapshot of the most recent documentation available at the time of its writing.

B Ontology Terms

B.1 Classes

This section contains a definition of all classes defined in the `fgo` namespace. For classes integrated from other ontologies, see the integration document 3.3.5.

Class: `fgo:Action`

label	Action
description	An action represents a specific routine which will be performed when a certain condition is fulfilled within a certain screen component. Examples are methods of a Web service (e.g., getItem) or functionality to update or change the contents of a form.
sub_class_of	`fgo:BuildingBlock`
in_range_of	`fgo:hasAction`

Class: `fgo:BackendService`

label	Backend Service
description	A Web service which provides data and/or functionality to a screen. The actual backend service is external to FAST, and only available through a wrapper (the service Resource).
sub_class_of	`fgo:BuildingBlock`

Class: `fgo:BuildingBlock`

label	BuildingBlock
description	Anything that is part of a gadget. Tentatively anything that can be 'touched' and moved around in the FAST IDE, from the most complex units such as screen flows, down to atomic form elements like a button or a label in a form.
super_class_of	`fgo:Action`, `fgo:BackendService`, `fgo:Condition`, `fgo:FormElement`, `fgo:Library`, `fgo:Pipe`, `fgo:Screen`, `fgo:ScreenComponent`, `fgo:ScreenFlow`, `fgo:Trigger`
in_domain_of	`fgo:contains`, `fgo:hasCopy`, `fgo:hasIcon`, `fgo:hasScreenshot`, `fgo:hasTemplate`, `fgo:uses`
in_range_of	`fgo:contains`, `fgo:hasCopy`, `fgo:hasTemplate`

Class: fgo:Condition

label	Condition
description	The pre- or post-condition of a certain kinds of building block. If the building block is a screen flow, each target platform will use these conditions in its own way, or may also ignore them. E.g., in EzWeb pre- and post-conditions correspond to the concepts of slot and event. A condition is made up of individual facts, where each fact is represented internally as an RDF triple, usually involving a variable of blank node.
sub_class_of	fgo:BuildingBlock
in_domain_of	fgo:hasPattern, fgo:hasPatternString, fgo:isPositive
in_range_of	fgo:from, fgo:hasPostCondition, fgo:hasPreCondition, fgo:to

Class: fgo:Form

label	Form
description	A form is the visual aspect of a screen: its user interface. Each form is made up of individual form elements.
sub_class_of	fgo:ScreenComponent
in_domain_of	fgo:hasFormElement
in_range_of	fgo:hasForm

Class: fgo:FormElement

label	Form Element
description	Form elements are UI elements in a particular screen, such as buttons, lists or labels.
sub_class_of	fgo:BuildingBlock
in_range_of	fgo:hasFormElement

Class: fgo:Library

label	Library
description	Libraries are references to external code libraries required for the execution of a particular building block at runtime.
sub_class_of	fgo:BuildingBlock
in_domain_of	fgo:hasLanguage
in_range_of	fgo:hasLibrary

Class: fgo:Operator

label	Operator
description	Operators are intended to transform and/or modify data within a screen, usually for preparing data coming from service resources for the use in the screen's interface. Operators cover different kinds of data manipulations, from simple aggregation to mediating data with incompatible schemas.
sub_class_of	fgo:ScreenComponent
in_range_of	fgo:hasOperator

Class: fgo:Pipe

label	pipe or connector
description	Pipes are used to explicitly define the flow of data within a screen, e.g., from service resource to operator to a specific form element.
sub_class_of	fgo:BuildingBlock
in_domain_of	fgo:from, fgo:to

Class: fgo:Resource

label	Resource
description	A service resource in FAST is a wrapper around a Web service (the backend service), which makes the service available to the platform, e.g., by mapping its definition to FAST facts and actions.
sub_class_of	fgo:ScreenComponent
in_range_of	fgo:hasResource

Class: fgo:Screen

label	Screen
description	An individual screen; the basic unit of user interaction in FAST. A screen is the interface through which a user gets access to data and functionality of a backend service.
sub_class_of	fgo:BuildingBlock
in_domain_of	fgo:hasForm, fgo:hasOperator, fgo:hasResource

Class: fgo:ScreenComponent

label	Screen Component
description	Screens are made up of screen components, which fundamentally include service resources, operators and forms.
super_class_of	fgo:Form, fgo:Operator, fgo:Resource
sub_class_of	fgo:BuildingBlock
in_domain_of	fgo:hasAction, fgo:hasTrigger

Class: fgo:ScreenFlow

label	Screen Flow
description	A set of screens from which a gadget for a given target platform can be generated.
sub_class_of	fgo:BuildingBlock

Class: fgo:Trigger

label	Trigger
description	Triggers are the flip-side of actions. Certain events in a building block can cause a trigger to be fired. Other building blocks within the same screen, which are listening to it, will react with an action.
sub_class_of	fgo:BuildingBlock
in_range_of	fgo:hasTrigger

Class: fgo:WithCode

label	With Code
description	Any kind of building block that can be defined as a whole through code.
in_domain_of	fgo:hasCode, fgo:hasLibrary
unionOf	fgo:Form, fgo:Operator, fgo:Resource, fgo:Screen

Class: fgo:WithDefinition

label	With Definition
description	Any kind of building block that can be defined declaratively in the GVS.
unionOf	fgo:Form, fgo:Screen

Class: `fgo:WithPostConditions`

label	With Post-condition
description	Those kinds of building blocks which can have post-conditions.
in_domain_of	`fgo:hasPostCondition`
unionOf	`fgo:Screen`, `fgo:ScreenComponent`, `fgo:ScreenFlow`

Class: `fgo:WithPreConditions`

label	With Pre-condition
description	Those kinds of building blocks which can have pre-conditions.
in_domain_of	`fgo:hasPreCondition`
unionOf	`fgo:Action`, `fgo:Screen`, `fgo:ScreenFlow`

B.2 Properties

This section contains a definition of all properties defined in the `fgo` namespace. For properties integrated from other ontologies, see the integration document 3.3.5.

Property: `fgo:contains`

label	contains
description	Many kinds of components in FAST can contain other components: screenflows contain screens, screens contain forms, operators and resources, forms contain form elements, etc.
type	`owl:ObjectProperty`
domain	`fgo:BuildingBlock`
range	`fgo:BuildingBlock`

Property: fgo:from

label	from
description	This property defines the starting point of a pipe within a screen.
type	owl:ObjectProperty
domain	fgo:Pipe
range	fgo:Condition

Property: fgo:hasAction

label	has action
description	This property indicates which actions are associated and can be perfomed within a screen component.
type	owl:ObjectProperty
domain	fgo:ScreenComponent
range	fgo:Action

Property: fgo:hasCode

label	has code
description	This property links to the executable code of a particular building block (currently screens or screen components).
type	owl:ObjectProperty
domain	fgo:WithCode
range	foaf:Document

Property: fgo:hasCopy

label	has copy
description	Links the template of a building block as used in the palette or catalogue of available building blocks to its copy, as used in a particular screen flow.
type	owl:ObjectProperty
domain	fgo:BuildingBlock
range	fgo:BuildingBlock

Property: fgo:hasForm

label	has form
description	If a screen is defined declaratively, this property links it to its form (i.e., its visual user interface).
type	owl:ObjectProperty
domain	fgo:Screen
range	fgo:Form

Property: fgo:hasFormElement

label	has form element
description	If a form is defined declaratively, its elements are linked to it with this property.
type	owl:ObjectProperty
domain	fgo:Form
range	fgo:FormElement

Property: fgo:hasIcon

label	has icon
description	A small graphical representation of any FAST component or sub-component.
type	owl:ObjectProperty
domain	fgo:BuildingBlock
range	foaf:Image

Property: fgo:hasLanguage

label	has language string
description	This property defines the programming language a particular programming library uses.
type	owl:DatatypeProperty
domain	fgo:Library
range	xsd:string

Property: fgo:hasLibrary

label	has library
description	This property indicates which programming libraries are used by the code of a screen component.
type	owl:ObjectProperty
domain	fgo:WithCode
range	fgo:Library

Property: fgo:hasOperator

label	has operator
description	If a screen is defined declaratively, this property links it to an operator it might contain.
type	owl:ObjectProperty
domain	fgo:Screen
range	fgo:Operator

Property: fgo:hasPattern

label	has pattern
description	This property links the abstract representation of a fact (i.e., an RDF triple) in FAST to a graph resource containing the actual triple.
type	owl:ObjectProperty
domain	fgo:Condition
range	rdfs:Resource

Property: fgo:hasPatternString

label	has pattern string
description	This property links to the textual representation of a fact (i.e., an RDF triple), e.g., in N3.
type	owl:DatatypeProperty
domain	fgo:Condition
range	xsd:String

Property: fgo:hasPostCondition

label	has post-condition
description	This property links certain type of building blocks to their post-condition, i.e., the facts that are produced once the building block has been executed.
type	owl:ObjectProperty
domain	fgo:WithPostConditions
range	fgo:Condition

Property: fgo:hasPreCondition

label	has pre-condition
description	This property links certain type of building blocks to their pre-condition, i.e., the facts that need to be fulfilled in order for the building block to be reachable.
type	owl:ObjectProperty
domain	fgo:WithPreConditions
range	fgo:Condition

Property: fgo:hasResource

label	has resource
description	If a screen is defined declaratively, this property links it to its service resource (i.e., its backend).
type	owl:ObjectProperty
domain	fgo:Screen
range	fgo:Resource

Property: fgo:hasScreenshot

label	has screenshot
description	An image which shows a particular screen or screenflow in action, to aid users in deciding which screen or screenflow to choose out of many.
type	owl:ObjectProperty
domain	fgo:BuildingBlock
range	foaf:Image

Property: `fgo:hasTemplate`

label	has template
description	Links the copy of a building block as used in a particular screen flow to its template, as used in the palette or catalogue of available building blocks.
type	`owl:ObjectProperty`
domain	`fgo:BuildingBlock`
range	`fgo:BuildingBlock`

Property: `fgo:hasTrigger`

label	has trigger string
description	This property links a buildiong block to a trigger.
type	`owl:DatatypeProperty`
domain	`fgo:ScreenComponent`
range	`fgo:Trigger`

Property: `fgo:isPositive`

label	is positive
description	Facts can be set to a specific scope: design time, execution time, or both of them. This property defines how they have to be taken into account by the inference engine or reasoner.
type	`owl:DatatypeProperty`
domain	`fgo:Condition`
range	`xsd:boolean`

Property: `fgo:to`

label	to
description	This property defines the end point of a pipe within a screen.
type	`owl:ObjectProperty`
domain	`fgo:Pipe`
range	`fgo:Condition`

Property: `fgo:uses`

label	uses
description	This property indicates concepts used within a building block, without being a pre- or post-condition.
type	`owl:ObjectProperty`
domain	`fgo:BuildingBlock`
range	`rdfs:Resource`

C Ontology Code

Below is the complete code of the FAST gadget ontology in Turtle syntax. The latest version of this file can always be accessed at `http://purl.oclc.org/fast/ontology/gadget`.

```
# FAST Gadget Ontology
#
# developed as part of the EU FAST Project (FP7-ICT-2007-1-216048)
#
# editor: Knud Hinnerk M ller, DERI, National Unversity of Ireland, Galway
# contributor: Ismael Rivera, DERI, National Unversity of Ireland, Galway
#
# this turtle file is the original document, from which all other versions
# (html, rdf/xml) are created
#
# $Id: fgo2011-02-11.ttl 24 2011-02-17 18:46:31Z knud $

@prefix fgo: <http://purl.oclc.org/fast/ontology/gadget#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>  .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix doap: <http://usefulinc.com/ns/doap#> .
@prefix sioc: <http://rdfs.org/sioc/ns#> .
@prefix dcterms: <http://purl.org/dc/terms/> .
@prefix misc: <http://data.semanticweb.org/ns/misc#> .
@prefix ctag: <http://commontag.org/ns#> .
@prefix swrc_ext: <http://www.cs.vu.nl/~mcaklein/onto/swrc_ext/2005/05#> .
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .
@prefix vs: <http://www.w3.org/2003/06/sw-vocab-status/ns#> .

<http://purl.oclc.org/fast/ontology/gadget> a owl:Ontology ;
    rdfs:label "The FAST Gagdet Ontology v0.3 (M36)"@en ;
    rdfs:comment """This ontology defines terms for modelling semantic, interoperable
        gadgets or widgets. It has been developed as part of the EU project 'FAST' (FAST
         AND ADVANCED STORYBOARD TOOLS), FP7-ICT-2007-1-216048."""@en ;
    foaf:logo <http://kantenwerk.org/ontology/fast_gadget_content/images/fast.png> ;
    dcterms:created "2009-02-09"@en ;
    dcterms:modified "$Date: 2011-02-17 18:46:31 +0000 (Thu, 17 Feb 2011) $"@en ;
    dcterms:creator <http://kantenwerk.org/metadata/foaf.rdf#me> ;
    foaf:maker <http://kantenwerk.org/metadata/foaf.rdf#me> ;
    swrc_ext:authorList fgo:author_list;
    dcterms:contributor fgo:Ismael , fgo:fast_members ;
    dcterms:license <http://creativecommons.org/licenses/by/3.0/>;
    misc:licenseDoc fgo:license_doc;
    foaf:maker <http://kantenwerk.org/metadata/foaf.rdf#me> ;
    vs:userdocs ( fgo:introduction fgo:basic_model ) ;
    doap:revision "$Revision: 24 $" ;
    sioc:earlier_version <http://kantenwerk.org/ontology/fast_gadget_content/fgo2009
        -11-16.rdf>;
    dcterms:hasVersion <http://kantenwerk.org/ontology/fast_gadget_content/fgo2009
        -11-16.rdf> .

<http://kantenwerk.org/metadata/foaf.rdf#me> a foaf:Person ;
    foaf:name "Knud M ller"@en ;
    foaf:homepage <http://kantenwerk.org> ;
    rdfs:seeAlso <http://kantenwerk.org/metadata/foaf.rdf> .

<http://data.semanticweb.org/organization/deri-nui-galway> a foaf:Organization;
    foaf:name "DERI"@en ;
    foaf:homepage <http://www.deri.ie> ;
    foaf:member <http://kantenwerk.org/metadata/foaf.rdf#me> ;
    foaf:member fgo:Ismael .
```

```
fgo:fast_members a foaf:Group ;
    foaf:name "Members of the FAST project"@en .

fgo:Ismael a foaf:Person ;
    foaf:name "Ismael Rivera"@en ;
    foaf:homepage <http://www.deri.ie/about/team/member/ismael_rivera/> ;
    rdfs:seeAlso <http://www.deri.ie/fileadmin/scripts/foaf.php?id=356> .

fgo:author_list a owl:Thing;
    rdf:_1 <http://kantenwerk.org/metadata/foaf.rdf#me> .

fgo:license_doc
    rdfs:label "License Statement";
    rdfs:comment """
            <!-- Creative Commons License -->
            <p class=\"copyright\">
                Copyright &copy; 2007-2011 FAST Project Consortium (FP7-ICT
                    -2007-1-216048).<br/>
                <br/>
                <a href=\"http://creativecommons.org/licenses/by/3.0/\"><img alt=\"
                    Creative Commons License\" style=\"border: 0; float: right; padding:
                     10px;\" src=\"images/somerights.gif\" /></a>
                This work is licensed under a <a href=\"http://creativecommons.org/
                    licenses/by/3.0/\">Creative Commons Attribution License</a>.
                This copyright applies to the <em>FAST Gadget Ontology (FGO)</em> and
                    accompanying documentation in RDF.
                Regarding underlying technology, FGO uses W3C's <a href=\"http://www.w3.
                    org/RDF/\">RDF</a> technology, an
                open Web standard that can be freely used by anyone.
            </p>

        """;
    a owl:Thing .

# documentation

fgo:introduction
    rdfs:label "Introduction"@en;
    rdfs:comment """
            <p>
                The EU <a href="http://fast.morfeo-project.org/" alt="FAST Project">FAST
                     Project</a> (FP7-ICT-2007-1-216048) sets out to develop
                end-user-focussed tools for building intelligent, interconnected Web
                    gadgets. In a visual development environment, users can
                drag &amp; drop building blocks, customise them and link them together
                    to create their own, custom-built gadget, which they can
                then deploy to a gadget platform like iGoogle or EZWeb.
            </p>
            <p class="centered"><a href="images/gvs-screenshot.png" alt="FAST GVS
                Screenshot"><img src="images/gvs-screenshot_thumb.png" title="FAST GVS
                Screenshot" class="doc_image"/></a></p>
            <p>
                Internally, each gadget and its building blocks are defined in RDF,
                    using the terminology defined by this document: the FAST
                gadget ontology (FGO). The <a href="#overview">overview</a> section
                    lists all classes and properties defined in the FGO. For a more
                detailed introduction to FAST (including a more extensive PDF version of
                     the FGO documentation), its architecture, software prototypes,
                etc., we would like to direct you to the collection of
                <a href="http://fast-fp7project.morfeo-project.org/documentation" alt="
                    FAST Documentation">FAST Documentation</a> documents.
            </p>
        """@en;
    a owl:Thing .

fgo:basic_model
    rdfs:label "The Basic Conceptual Model"@en ;
```

```
rdfs:comment """
        <p>
            One of the main ideas of FAST is that each gadget consists of so-called
                <strong>screens</strong>: individual units of interaction, which
            the user of the gadgets visits one after the other. E.g., in a gadget
                for buying something in an online store, there could be a login
                screen,
            followed by a search screen, a selection screen and finally a check-out
                screen. Each screen has its own building blocks, such as a
            <strong>form</strong> or a any number of <strong>operators</strong>. In
                order for the screens to interact with services like the
            online store, they are linked to <strong>resources</strong>, which are
                basically wrappers around backend web services. Several screens
            together build a so-called <strong>screenflow</strong>, which can then
                be packaged and deployed as the actual gadget the end user will see.
            The layering of all these basic concepts can be seen in the figure below
                .
        </p>
        <p class="centered"><a href="images/uml_composition.png" alt="Layering of
            basic FAST concepts"><img src="images/uml_composition_thumb.png" title="
            Layering of basic FAST concepts" class="doc_image"/></a></p>
        <p>
            The next figure looks closer at the features of individual screens.
                These are <strong>forms</strong>, <strong>resources</strong> and
            <strong>operators</strong>, which are all specialisations
            of the general <strong>screen_component</strong> concept. Each such
                screen component may or may not have an <strong>action</strong> or
            <strong>trigger</strong> associated with them, which declaratively
                define their behaviour in terms of their own functionality or
                functionality
            they can trigger in other building blocks.
        </p>
        <p class="centered"><a href="images/uml_screen_component.png" alt="FAST
            Screen Components"><img src="images/uml_screen_component_thumb.png"
            title="FAST Screen Components" class="doc_image"/></a></p>
        <p>
            Regarding their implementation, different kinds of building blocks in
                FAST can either be defined hard-coded as a piece of source code, or
                they
            can be defined declaratively in the terms of the FAST ontology (using
                pipes, operators, etc.). In the former case, a building block such
                as a
            screen would have been implemented and added to the FAST platform by an
                engineer, whereas in the latter case, ordinary users of the FAST
                platform
            would have assembled the building block using the tools available in the
                 GVS. This principle is reflected in the figure below, showing how
                certain
            kinds of building blocks aggregate either a <strong>Definition</strong>
                or <strong>Code</strong>, whereas other kinds of building blocks are
                 always
            hard-coded. It should be noted that, in theory, there is no reason why
                not any kind of building block could be defined either way. However,
                 the
            concrete set of building blocks which can be defined declaratively or in
                 code is changing dynamically according to the current state and
                plans of
            the FAST project.
        </p>
        <p class="centered"><a href="images/uml_definition.png" alt="Defining
            Screens in FAST"><img src="images/uml_definition.png" title="Defining
            Screens in FAST" class="doc_image"/></a></p>
        <p>
            Another central concept in FAST is the idea that, rather than explicitly
                 defining the order in which screens are shown to the user in a
                gadget,
            the order is determined automatically. This happens through the use of <
                strong>pre-conditions</strong> and <strong>post-conditions</strong>:
```

```
                a screen will only show up once all its pre-conditions are fulfilled,
                    and it will produce post-conditions for other screens once the user
                    is
                done with it. This idea is extended from screens to other kinds of
                    building blocks in FAST, to determine the general flow of data.
                The figure below illustrates how the different kinds of building blocks
                    relate to the concepts of pre- and post-conditions. While all
                building blocks can essentially have such conditions, only screens and
                    screen flows directly aggregate both. However, in the case of the
                    three
                different kinds of screen components (<strong>form</strong>, <strong>
                    resource</strong> and <strong>operator</strong>), the relation to a
                pre-condition is only established through their actions, which represent
                     their basic functionality. For all building blocks, both
                pre- and post-conditions are entirely optional.
            </p>
            <p class="centered"><a href="images/uml_condition.png" alt="FAST building
                blocks with pre- and post-conditions"><img src="images/
                uml_condition_thumb.png" title="FAST building blocks with pre- and post-
                conditions" class="doc_image"/></a></p>
    """@en ;
    a owl:Thing .

# some topics:

fgo:TopicBuildingBlocks a skos:Concept ;
    rdfs:label "FAST Building Blocks"@en ;
    skos:prefLabel "FAST Building Blocks"@en .

fgo:TopicConditions a skos:Concept ;
    rdfs:label "Conditions"@en ;
    skos:prefLabel "Conditions"@en .

fgo:TopicDefiningBuildingBlocks a skos:Concept ;
    rdfs:label "Defining Building Blocks"@en ;
    skos:prefLabel "Defining Building Blocks"@en .

# Classes

### Building Blocks (Screenflows, Screens and screen components)

fgo:BuildingBlock a owl:Class ;
    rdfs:label "BuildingBlock"@en ;
    rdfs:comment """Anything that is part of a gadget. Tentatively anything that can be
        'touched' and moved around in the FAST IDE, from the most complex units such as
        screen flows, down to atomic form elements like a button or a label in a form.""
        "@en ;
    dcterms:subject fgo:TopicBuildingBlocks .

fgo:ScreenFlow a owl:Class ;
    rdfs:subClassOf fgo:BuildingBlock ;
    rdfs:label "Screen Flow"@en ;
    rdfs:comment "A set of screens from which a gadget for a given target platform can
        be generated."@en ;
    dcterms:subject fgo:TopicBuildingBlocks .

fgo:Screen a owl:Class ;
rdfs:subClassOf fgo:BuildingBlock ;
rdfs:label "Screen"@en ;
rdfs:comment "An individual screen; the basic unit of user interaction in FAST. A screen
     is the interface through which a user gets access to data and functionality of a
    backend service."@en   ;
    dcterms:subject fgo:TopicBuildingBlocks .

fgo:ScreenComponent a owl:Class ;
rdfs:subClassOf fgo:BuildingBlock ;
rdfs:label "Screen Component"@en ;
```

```
rdfs:comment "Screens are made up of screen components, which fundamentally include
    service resources, operators and forms."@en  ;
    dcterms:subject fgo:TopicBuildingBlocks .

fgo:Operator a owl:Class ;
rdfs:subClassOf fgo:ScreenComponent ;
rdfs:label "Operator"@en ;
rdfs:comment """Operators are intended to transform and/or modify data within a screen,
    usually for preparing data coming from service resources for the use in the screen's
     interface. Operators cover different kinds of data manipulations, from simple
    aggregation to mediating data with incompatible schemas."""@en ;
    dcterms:subject fgo:TopicBuildingBlocks .

fgo:Resource a owl:Class ;
rdfs:subClassOf fgo:ScreenComponent ;
rdfs:label "Resource"@en ;
rdfs:comment "A service resource in FAST is a wrapper around a Web service (the backend
    service), which makes the service available to the platform, e.g., by mapping its
    definition to FAST facts and actions."@en ;
    dcterms:subject fgo:TopicBuildingBlocks .

fgo:Form a owl:Class ;
rdfs:subClassOf fgo:ScreenComponent ;
rdfs:label "Form"@en ;
rdfs:comment "A form is the visual aspect of a screen: its user interface. Each form is
    made up of individual form elements."@en ;
    dcterms:subject fgo:TopicBuildingBlocks .

fgo:BackendService a owl:Class ;
rdfs:subClassOf fgo:BuildingBlock ;
rdfs:label "Backend Service"@en ;
rdfs:comment """A Web service which provides data and/or functionality to a screen. The
    actual backend service is external to FAST, and only available through a wrapper (
    the service Resource)."""@en  ;
    dcterms:subject fgo:TopicBuildingBlocks .

fgo:FormElement a owl:Class ;
rdfs:subClassOf fgo:BuildingBlock ;
rdfs:label "Form Element"@en ;
rdfs:comment "Form elements are UI elements in a particular screen, such as buttons,
    lists or labels."@en ;
    dcterms:subject fgo:TopicBuildingBlocks .

### Conditions

fgo:Condition a owl:Class ;
rdfs:subClassOf fgo:BuildingBlock ;
rdfs:label "Condition"@en ;
rdfs:comment """The pre- or post-condition of a building block. If the building block is
     a screen flow, each target platform will use these conditions in its own way, or
    may also ignore them. E.g., in EzWeb pre- and post-conditions correspond to the
    concepts of slot and event.
A condition is made up of individual facts, where each fact is represented internally as
     an RDF triple, usually involving a variable of blank node."""@en ;
   dcterms:subject fgo:TopicConditions .


fgo:WithPreConditions a owl:Class ;
owl:unionOf (fgo:ScreenFlow fgo:Screen fgo:Action) ;
rdfs:label "With Pre-condition"@en ;
rdfs:comment """Those kinds of building blocks which can have pre-conditions."""@en ;
    dcterms:subject fgo:TopicConditions .

fgo:WithPostConditions a owl:Class ;
owl:unionOf (fgo:ScreenFlow fgo:Screen fgo:ScreenComponent) ;
rdfs:label "With Post-condition"@en ;
rdfs:comment """Those kinds of building blocks which can have post-conditions."""@en ;
    dcterms:subject fgo:TopicConditions .
```

```
# defining building blocks

fgo:WithDefinition a owl:Class ;
owl:unionOf (fgo:Screen fgo:Form) ;
rdfs:label "With Definition"@en ;
rdfs:comment "Any kind of building block that can be defined declaratively in the GVS."
    @en ;
    dcterms:subject fgo:TopicDefiningBuildingBlocks .

fgo:WithCode a owl:Class ;
owl:unionOf (fgo:Screen fgo:Form fgo:Resource fgo:Operator) ;
rdfs:label "With Code"@en ;
rdfs:comment "Any kind of building block that can be defined as a whole through code."
    @en ;
    dcterms:subject fgo:TopicDefiningBuildingBlocks .

fgo:Action a owl:Class ;
rdfs:subClassOf fgo:BuildingBlock ;
rdfs:label "Action"@en ;
rdfs:comment """An action represents a specific routine which will be performed when a
    certain
condition is fulfilled within a certain screen component. Examples are methods of a Web
    service (e.g., getItem) or functionality to update or change the contents of a form.
    """@en;
    dcterms:subject fgo:TopicDefiningBuildingBlocks .

fgo:Library a owl:Class ;
rdfs:subClassOf fgo:BuildingBlock ;
rdfs:label "Library"@en ;
rdfs:comment """Libraries are references to external code libraries required for the
    execution of a particular building block at runtime."""@en ;
    dcterms:subject fgo:TopicDefiningBuildingBlocks .

fgo:Pipe a owl:Class ;
rdfs:subClassOf fgo:BuildingBlock ;
rdfs:label "pipe or connector"@en ;
rdfs:comment """Pipes are used to explicitly define the flow of data within a screen, e.
    g., from service resource to operator to a specific form element."""@en ;
    dcterms:subject fgo:TopicDefiningBuildingBlocks .

fgo:Trigger a owl:Class ;
rdfs:subClassOf fgo:BuildingBlock ;
rdfs:label "Trigger"@en ;
rdfs:comment """Triggers are the flip-side of actions. Certain events in a building
    block can cause a trigger to be fired. Other building blocks within the same screen,
     which are listening to it, will react with an action."""@en ;
    dcterms:subject fgo:TopicDefiningBuildingBlocks .


# Properties

fgo:contains a owl:ObjectProperty ;
rdfs:label "contains"@en ;
rdfs:comment """Many kinds of components in FAST can contain other components:
    screenflows contain screens, screens contain forms, operators and resources, forms
    contain form elements, etc."""@en ;
rdfs:domain fgo:BuildingBlock ;
rdfs:range fgo:BuildingBlock .

fgo:hasForm a owl:ObjectProperty ;
rdfs:subPropertyOf fgo:contains ;
rdfs:label "has form"@en ;
rdfs:comment "If a screen is defined declaratively, this property links it to its form (
    i.e., its visual user interface)."@en ;
rdfs:domain fgo:Screen ;
rdfs:range fgo:Form .
```

```
fgo:hasResource a owl:ObjectProperty ;
rdfs:subPropertyOf fgo:contains ;
rdfs:label "has resource"@en ;
rdfs:comment "If a screen is defined declaratively, this property links it to its
    service resource (i.e., its backend)."@en ;
rdfs:domain fgo:Screen ;
rdfs:range fgo:Resource .

fgo:hasOperator a owl:ObjectProperty ;
rdfs:subPropertyOf fgo:contains ;
rdfs:label "has operator"@en ;
rdfs:comment "If a screen is defined declaratively, this property links it to an
    operator it might contain."@en ;
rdfs:domain fgo:Screen ;
rdfs:range fgo:Operator .

fgo:hasFormElement a owl:ObjectProperty ;
rdfs:subPropertyOf fgo:contains ;
rdfs:label "has form element"@en ;
rdfs:comment "If a form is defined declaratively, its elements are linked to it with
    this property."@en ;
rdfs:domain fgo:Form ;
rdfs:range fgo:FormElement .

fgo:hasPreCondition a owl:ObjectProperty ;
rdfs:label "has pre-condition"@en ;
rdfs:comment """This property links certain type of building blocks to their pre-
    condition, i.e., the facts that need to be fulfilled in order for the building block
     to be reachable."""@en ;
rdfs:domain fgo:WithPreConditions ;
rdfs:range fgo:Condition .

fgo:hasPostCondition a owl:ObjectProperty ;
rdfs:label "has post-condition"@en ;
rdfs:comment """This property links certain type of building blocks to their post-
    condition, i.e., the facts that are produced once the building block has been
    executed."""@en ;
rdfs:domain fgo:WithPostConditions ;
rdfs:range fgo:Condition .

fgo:hasAction a owl:ObjectProperty ;
rdfs:label "has action"@en ;
rdfs:comment """This property indicates which actions are associated and can be perfomed
     within a screen component."""@en ;
rdfs:domain fgo:ScreenComponent ;
rdfs:range fgo:Action .

fgo:hasTrigger a owl:DatatypeProperty ;
rdfs:label "has trigger string"@en ;
rdfs:comment """This property links a buildiong block to a trigger."""@en ;
rdfs:domain fgo:ScreenComponent ;
rdfs:range fgo:Trigger .

fgo:uses a owl:ObjectProperty ;
rdfs:label "uses"@en ;
rdfs:comment """This property indicates concepts used within a building block, without
    being a pre- or post-condition."""@en ;
rdfs:domain fgo:BuildingBlock ;
rdfs:range rdfs:Resource .

# Defining Condtions:

fgo:hasPattern a owl:ObjectProperty ;
rdfs:label "has pattern"@en ;
rdfs:comment """This property links the abstract representation of a fact (i.e., an RDF
    triple) in FAST to a graph resource containing the actual triple."""@en ;
rdfs:domain fgo:Condition ;
rdfs:range rdfs:Resource .
```

```
fgo:hasPatternString a owl:DatatypeProperty ;
rdfs:label "has pattern string"@en ;
rdfs:comment """This property links to the textual representation of a fact (i.e., an
    RDF triple), e.g., in N3."""@en ;
rdfs:domain fgo:Condition ;
rdfs:range xsd:String .

fgo:isPositive a owl:DatatypeProperty ;
rdfs:label "is positive"@en ;
rdfs:comment """Facts can be set to a specific scope: design time, execution time, or
    both of them. This property defines how they have to be taken into account by the
    inference engine or reasoner."""@en ;
rdfs:domain fgo:Condition ;
rdfs:range xsd:boolean .

# annotating building blocks

fgo:hasIcon a owl:ObjectProperty ;
rdfs:label "has icon"@en ;
rdfs:comment "A small graphical representation of any FAST component or sub-component."
    @en ;
rdfs:subPropertyOf foaf:depiction ;
rdfs:domain fgo:BuildingBlock ;
rdfs:range foaf:Image .

fgo:hasScreenshot a owl:ObjectProperty ;
rdfs:label "has screenshot"@en ;
rdfs:comment """An image which shows a particular screen or screenflow in action, to aid
     users in deciding which screen or screenflow to choose out of many."""@en ;
rdfs:subPropertyOf foaf:depiction ;
rdfs:domain fgo:BuildingBlock ;
rdfs:range foaf:Image .


# properties concerning definitions through code

fgo:hasCode a owl:ObjectProperty ;
rdfs:label "has code"@en ;
rdfs:comment """This property links to the executable code of a particular building
    block (currently screens or screen components)."""@en ;
rdfs:domain fgo:WithCode ;
rdfs:range foaf:Document .

fgo:hasLibrary a owl:ObjectProperty ;
rdfs:label "has library"@en ;
rdfs:comment """This property indicates which programming libraries are used by the code
     of a screen component."""@en ;
rdfs:domain fgo:WithCode ;
rdfs:range fgo:Library .

fgo:hasLanguage a owl:DatatypeProperty ;
rdfs:label "has language string"@en ;
rdfs:comment """This property defines the programming language a particular programming
    library uses."""@en ;
rdfs:domain fgo:Library ;
rdfs:range xsd:string .


# relating resource templates and copies:

fgo:hasPrototype a owl:ObjectProperty ;
rdfs:label "has prototype"@en ;
rdfs:comment "Links the clone of a building block as used in a particular screen flow or
     screen to its prototype, as used in the palette or catalogue of available building
    blocks."@en ;
rdfs:domain fgo:BuildingBlock ;
```

```
rdfs:range fgo:BuildingBlock ;
owl:inverseOf fgo:hasClone .

fgo:hasClone a owl:ObjectProperty ;
rdfs:label "has copy"@en ;
rdfs:comment "Links the property of a building block as used in the palette or catalogue
     of available building blocks to its clone, as used in a particular screen flow or
    screen."@en ;
rdfs:domain fgo:BuildingBlock ;
rdfs:range fgo:BuildingBlock ;
owl:inverseOf fgo:hasPrototype .

# defining pipes

fgo:from a owl:ObjectProperty ;
rdfs:label "from"@en ;
rdfs:comment "This property defines the starting point of a pipe within a screen." ;
rdfs:domain fgo:Pipe ;
rdfs:range fgo:Condition .

fgo:to a owl:ObjectProperty ;
rdfs:label "to"@en ;
rdfs:comment "This property defines the end point of a pipe within a screen." ;
rdfs:domain fgo:Pipe ;
rdfs:range fgo:Condition .

# terms used from other ontologies

misc:integratesTerm a owl:ObjectProperty ;
rdfs:label "integratesTerm"@en ;
rdfs:comment """A way to explicitly say that an ontology uses terms from another
    namespace."""@en .

# this partially reflects the integration document of the FAST gadget ontology

<http://purl.oclc.org/fast/ontology/gadget> misc:integratesTerm
foaf:Person, foaf:Image, foaf:OnlineAccount,
foaf:holdsAccount, foaf:depiction, foaf:name, foaf:mbox, foaf:mbox_sha1sum, foaf:
    interest,
foaf:accountName,
sioc:User,
ctag:Tag,
ctag:tagged, ctag:means,
dcterms:title, dcterms:creator, dcterms:description, dcterms:created, dcterms:subject,
dcterms:rights, dcterms:RightsStatement, dcterms:rightsHolder,
doap:revision .

# statements to ensure that the ontology stays in DL
# according to Pellet reasoner: http://www.mindswap.org/2003/pellet/demo

foaf:Person a owl:Class .
foaf:Organization a owl:Class .
foaf:Image a owl:Class .
foaf:OnlineAccount a owl:Class .
sioc:User a owl:Class .
dcterms:RightsStatement a owl:Class .
ctag:Tag a owl:Class .

dcterms:modified a owl:DatatypeProperty .
dcterms:created a owl:DatatypeProperty .
dcterms:creator a owl:ObjectProperty .
dcterms:title a owl:DatatypeProperty .
dcterms:description a owl:DatatypeProperty .
dcterms:subject a owl:ObjectProperty .
dcterms:rights a owl:ObjectProperty .
dcterms:rightsHolder a owl:ObjectProperty .

foaf:name a owl:DatatypeProperty .
```

```
foaf:accountName a owl:DatatypeProperty .
foaf:holdsAccount a owl:ObjectProperty .
foaf:depiction a owl:ObjectProperty .
foaf:mbox a owl:ObjectProperty .
foaf:mbox_sha1sum a owl:DatatypeProperty .
foaf:interest a owl:ObjectProperty .
foaf:member a owl:ObjectProperty .

doap:revision a owl:DatatypeProperty .

ctag:tagged a owl:ObjectProperty .
ctag:means a owl:ObjectProperty .

<http://purl.oclc.org/fast/ontology/gadget> a owl:Thing.
```

D The VocDoc Ontology Documentation Tool

VocDoc is a tool for the auto-generation of documentation for RDF vocabularies and ontologies. VocDoc was originally developed for use within the FAST project, but has since been used in other contexts as well, such as for generating the online documentation of the Semantic Web Conference Ontology[16]. As input, VocDoc parses the original RDF source files of the ontology, and then produces documentation in different output formats, such as HTML for online purposes, DOT[17] for a visual overview and LaTeX for print purposes.

VocDoc is available at `http://kantenwerk.org/vocdoc`.

D.1 VocDoc Output

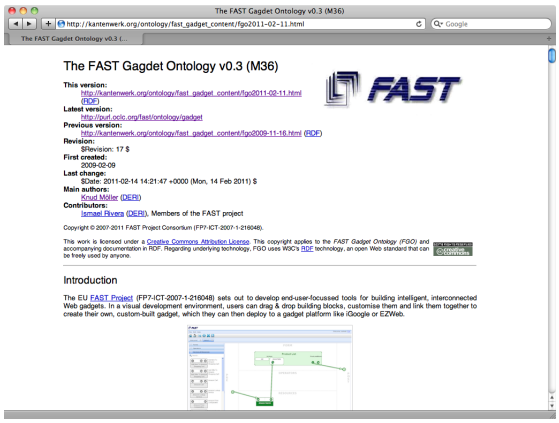
At the moment, VocDoc will either produce LaTeX, DOT or HTML output, depending on the parameters specified when running the tool (see Sect. D.3.1).

The LaTeX output is currently intended for inclusion in other documents. It consists of a list ontology terms, divided into classes and properties, with a detailed description for each. These descriptions include relations to other ontology terms (hierarchical or other), as well as various other attributes available through OWL and RDFS semantics. Related terms are linked to each other via hyperlinks. App. B of this deliverable was created entirely via VocDoc and serves as an example of this kind of output.

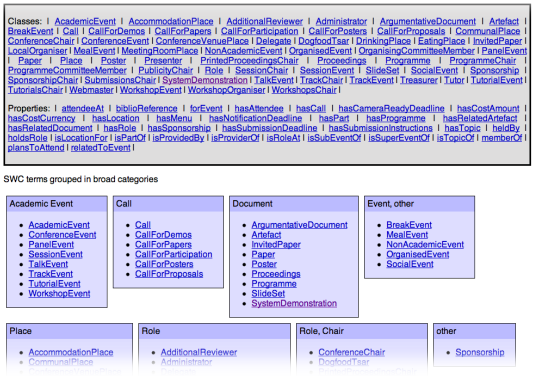
The DOT output of VocDoc will produce a UML class hierarchy of the input ontology's classes in the DOT language. The ontology overview in Fig. 8 was produced in this fashion.

The HTML option of VocDoc produces a stand-alone HTML document for publishing on the Web. It includes the same interlinked term list as the LaTeX output. In addition, and if the appropriate annotations are present in the input graph (see Sect. D.2), the output document will also include a preamble with metadata such as authors and change date, licensing information, an overview with terms group by topic, and extended additional documentation.

[16] `http://data.semanticweb.org/ns/swc/ontology`
[17] `http://www.graphviz.org/doc/info/lang.html`



(a) Preamble of the HTML output



(b) Overview section of the HTML output

Figure 12: Screenshots showing the HTML output of VocDoc

D.2 Configuring VocDoc Processing

In principle, VocDoc will parse any input RDF and produce output accordingly, scanning the graph for T-Box terms from the `rdf`[18], `rdfs`[19] and `owl`[20] namespaces such as class and property definitions, class hierarchies or domain and range specifications. However, the output of VocDoc can be further manipulated and improved by using specific annotation statements in the input graph, providing a configuration file and using command line parameters, as explained in the sections below.

D.2.1 T-Box Definitions and Annotations

VocDoc will scan the input graph for instances of the following classes from the `owl` namespace:

- **Classes**: `owl:Class` and `owl:DeprecatedClass`
- **Properties**: `owl:ObjectProperty`, `owl:DatatypeProperty`, `owl:FunctionalProperty`, `owl:InverseFunctionalProperty`, `owl:TransitiveProperty`, `owl:SymmetricProperty` and `owl:DeprecatedProperty`
- **Individuals**: instances of any of the classes defined in this ontology

[18] `http://www.w3.org/1999/02/22-rdf-syntax-ns#`
[19] `http://www.w3.org/2000/01/rdf-schema#`
[20] `http://www.w3.org/2002/07/owl#`

For each class, the tool will furthermore find the class hierarchy (via `rdfs:subClassOf`), as well as class definitions based on OWL constructs such as `owl:unionOf`, `owl:intersectionOf` and `owl:oneOf`. Additionally, it is checked if the class is in the `rdfs:domain` or `rdfs:range` of any property defined in the same ontology.

Each property is checked for its domain and range `rdfs:range` and `rdfs:domain`, placed in a term hierarchy using `rdfs:subPropertyOf` and `rdfs:superPropertyOf` and checked for `owl:inverseOf` statements.

All terms in the ontology (classes and properties) can also have further annotations, such as `rdfs:label`, `rdfs:comment` (the documentation for an individual term), `dc:subject` (for grouping ontology terms in different thematic categories) and `vs:term_status`, which can have values such as “testing” or “stable”.

D.2.2 Ontology Annotations

VocDoc looks for a number of special annotation properties to structure and enhance its output. All of these properties must be used on the main `owl:Ontology` instance in the input graph. Examples for these annotations can be found in App. C.

Title and Logo `rdfs:label` is used to specify the name of the vocabulary, which will appear as the ontology title in the output document. `foaf:logo` is used to specify the URL of a logo image, which will also be added to the output.

Dates The Dublin Core vocabulary is used to specify relevant dates in the ontology development process. Specifically, `dcterms:created` indicates the date when the vocabulary was first created, while `dcterms:modified` indicates the date when the vocabulary was last changed (i.e., this would always be the date when the current version of the ontology was created).

Authors `dcterms:creator` is used to specify the main authors of the vocabulary or ontology. There can be several creators, but each one must be an instance of `foaf:Person`, complete with a `foaf:name`, `foaf:surname` and `foaf:homepage`. Optionally, it is also possible to instantiate a `foaf:Organization` with name and homepage and say that the person is

foaf:member of it. Additional contributors to the vocabulary can be added can be added using dcterms:contributor. Details for contributors are defined in the same way as authors. However, it is also possible to have any kind of foaf:Agent as the value. This way, it is possible to have a group as a contributor, e.g., "members of the FAST project".

Licensing Information Human-readable licensing information can be added using the property dc:license to point to a resource which contains human-readable licensing information. The license resource itself should be modelled with an rdfs:label (e.g. "License Statement") and an rdfs:comment containing the actual license text as it should appear in the output document. rdfs:seeAlso can be used to point to a detailed version of the license (e.g., http://creativecommons.org/licenses/by-nc/3.0/).

General Documentation An arbitrary number of sections containing further human-readable documentation can be added. Each section is defined as an individual RDF resource (of arbitrary type), using rdfs:label (for the section's title) and rdfs:comment (for the section's content). These sections are linked to the ontology in the desired order using the vs:userDocs property. The object of the userDocs statement must be a list construct with the sections in the desired order.

D.2.3 Configuration File

VocDoc requires the presence of a configuration file, which is used to provide the tool with additional data required for processing the input ontology. The configuration file consists of the following simple key-value pairs (key=value):

- ontology_namespace: The namespace URI in which the ontology resides. This is used to disambiguate between different instances of owl:Ontology. This parameter is mandatory.
- doc_root: If HTML output is produced, this URI indicates where the ontology documentation and source will eventually be located online. This parameter is mandatory.
- alphabetise: A boolean which sets if term lists should be outputted in alphabetical order.

- `GA_code`: If HTML output is produced, VocDoc can append special code needed to make Google Analytics[21] work with the documentation. The code is the application code provided by Google for the user.

D.3 Running VocDoc

VocDoc is implemented in the Ruby scripting language as a command line tool, using the Redland RDF API[22] for parsing and querying the input RDF graph. The general syntax for running as shown below, where `source_file` indicates the input RDF file (in Turtle syntax) and `target_folder` indicates the folder in which to write the output LaTeX or HTML files.

```
ruby vocdoc.rb [parameters] source_file target_folder
```

D.3.1 Command-line Parameters

The following command-line parameters are available:

```
-h, --help          Displays help message
-v, --version       Display the version, then exit
-q, --quiet         Output as little as possible, overrides verbose
-V, --verbose       Verbose output
--latex             output latex source into target folder, divided
                      into "classes.tex" and "properties.tex". This
                      is the default.
--html              output html source ("ontology.html") into
                      target folder
```

D.3.2 Requirements

The following requirements must be fulfilled in order to run VocDoc:

- Ruby must be installed.

[21] http://www.google.com/analytics/
[22] http://librdf.org/

- The Redland RDF library and Ruby language bindings must be installed.
- A configuration file `owl_doc.conf` must be present in the folder where VocDoc is run.

E Past Changes

For keeping a record of past changes, this section simply lists the changes that were made between the previous two versions of this deliverable, D2.2.1 [Möller, 2009] and D2.2.2 [Möller, 2010].

- Sect. 1.1 has been revised to better define the scope of this deliverable with respect to other deliverables.
- The list of classes and properties has been moved to the appendix section at the back of the document (App. B), because it was felt that it would otherwise hinder the flow of the text.
- In response to the M12 review, numerous graphical representations both of the underlying conceptual model as well as the concrete gadget ontology have been added to the document, using UML notation. As an example, Sect. 4.1 now provides a complete overview of the ontology with all classes and properties.
- Also in response to the previous review, conceptual model and ontology have been extended both in breadth and depth.
- As an addition to the glossary of terms in the conceptualisation phase, a section discussing and illustrating the relations between the different classes has been provided.
- In order to reflect the extension of the scope of FAST towards screen creation (as opposed to screen flow creation), the conceptual model and ontology have been extended accordingly.
- A mechanism for RDF templating has been devised and is discussed in Sect. 4.2.
- As the result of the ongoing dialogue between ontology developers and application implementers, many classes and properties have been added, changed or removed.
- The section on our adopted development methodology (previously Sect. 1.3) has been moved to the end of the document (App. A.1), but otherwise did not change significantly.
- An evaluation section has been added (Sect. 5), based on research done by the authors within the past year.
- The section on design principles (previously Sect. 1.4, now Sect. 5.1) has been moved to the general evaluation section.

- In the integration section (Sect. 3.3) of the ontology definition, the Common Tag vocabulary for complex tagging has been added. It is briefly introduced, and the integrated terms are specified.
- On a technical note, the namespace abbreviation for the FAST gadget ontology changed from `fast` to `fgo`, in order to distinguish from other FAST ontologies which may be added in the future.

References

[Ambrus, 2010] Ambrus, O. (2010). Mediation amongst ontologies: Application to the FAST ontology. Deliverable D2.4.2, FAST Project (FP7-ICT-2007-1-216048).

[Ambrus et al., 2009] Ambrus, O., Möller, K., and Handschuh, S. (2009). Towards ontology matching for intelligent gadgets. In *Workshop on User-generated Services (UGS2009) at ICSOC2009, Stockholm, Sweden*.

[Auer et al., 2007] Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., and Ives, Z. (2007). DBpedia: A nucleus for a web of open data. In *6th International Semantic Web Conference and 2nd Asian Semantic Web Conference (ISWC+ASWC2007), Busan, South Korea*, pages 11–15. Springer.

[Berners-Lee, 2006] Berners-Lee, T. (2006). Linked data. `http://www.w3.org/DesignIssues/LinkedData.html`.

[Berrueta and Phipps, 2008] Berrueta, D. and Phipps, J. (2008). Best practice recipes for publishing RDF vocabularies. Working group note, W3C. `http://www.w3.org/TR/swbp-vocab-pub/` 06/05/2009.

[Bhiri, 2010] Bhiri, S. (2010). (Yet Another) Business Function Model. Internal presentation.

[Bizer and Cyganiak, 2004] Bizer, C. and Cyganiak, R. (2004). The TriG syntax. Technical report, Freie Universität Berlin. `http://www4.wiwiss.fu-berlin.de/bizer/TriG/` 22/09/2009.

[Breslin and Bojārs, 2009] Breslin, J. and Bojārs, U. (2009). SIOC core ontology specification. `http://rdfs.org/sioc/spec/`.

[Breslin et al., 2007] Breslin, J., Bojārs, U., Passant, A., and Polleres, A. (2007). SIOC ontology: Related ontologies and RDF vocabularies. `http://www.w3.org/Submission/sioc-related/`.

[Breslin et al., 2005] Breslin, J. G., Harth, A., Bojārs, U., and Decker, S. (2005). Towards Semantically-Interlinked Online Communities. In *The 2nd European Semantic Web Conference (ESWC '05), Heraklion, Greece, Proceedings, LNCS 3532*, pages 500–514.

[Brickley and Miller, 2007] Brickley, D. and Miller, L. (2007). FOAF Vocabulary Specification. `http://xmlns.com/foaf/0.1`.

[Davis, 2003] Davis, I. (2003). RDF Template Language 1.0. Draft, Semantic Planet. `http://www.semanticplanet.com/2003/08/rdft/spec` 28/01/2010.

[DCMI Usage Board, 2008] DCMI Usage Board (2008). DCMI metadata terms. `http://dublincore.org/documents/dcmi-terms/`.

[Fernández et al., 1997] Fernández, M., Gómez-Pérez, A., and Juristo, N. (1997). METHONTOLOGY: From ontological art towards ontological engineering. In *Workshop on Ontological Engineering at AAAI97, Stanford, USA*, Spring Symposium Series.

[Gómez-Pérez et al., 1996] Gómez-Pérez, A., Fernández, M., and de Vicente, A. J. (1996). Towards a method to conceptualize domain ontologies. In *Workshop on Ontological Engineering at ECAI'96*, pages 41–51.

[Hepp, 2008] Hepp, M. (2008). Goodrelations: An ontology for describing products and services offers on the web. In *16th International Conference on Knowledge Engineering and Knowledge Management (EKAW2008), Acitrezza, Italy*, volume 5268 of *Lecture Notes in Computer Science (LNCS)*, pages 329–346.

[Kawamoto et al., 2006] Kawamoto, K., Kitamura, Y., and Tijerino, Y. (2006). Kawawiki: A semantic wiki based on RDF templates. In *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology, Workshops (WI-IAT 2006 Workshops)*, pages 425–432. IEEE Computer Society.

[Lizcano et al., 2008] Lizcano, D., Soriano, J., Reyes, M., and Hierro, J. J. (2008). EzWeb/FAST: Reporting on a successful mashup-based solution for developing and deploying composite applications in the upcoming web of services. In *Proceedings of the 10th International Conference on Information Integration and Web-based Applications Services, iiWAS 2008, Linz, Austria*. ACM Press.

[Möller, 2009] Möller, K. (2009). Ontology and conceptual model for the semantic characterisation of complex gadgets. Deliverable D2.2.1, FAST Project (FP7-ICT-2007-1-216048).

[Möller, 2010] Möller, K. (2010). Ontology and conceptual model for the semantic characterisation of complex gadgets. Deliverable D2.2.2, FAST Project (FP7-ICT-2007-1-216048).

[Möller et al., 2010] Möller, K., Bechhofer, S., Heath, T., and Handschuh, S. (2010). Ontology soft skills — the Semantic Web Conference Ontology. *Applied Ontology*, Special Edition on "Beautiful Ontologies". (to appear).

[Möller et al., 2007] Möller, K., Heath, T., Handschuh, S., and Domingue, J. (2007). Recipes for Semantic Web dog food — the ESWC2006 and ISWC2006 metadata projects. In *6th International Semantic Web Conference and 2nd Asian Semantic Web Conference (ISWC+ASWC2007), Busan, South Korea*, pages 802–815. Springer.

[Palaghita and Rivera, 2010] Palaghita, C. and Rivera, I. (2010). User manual of the FAST catalogue. Deliverable D5.1.1, FAST Project (FP7-ICT-2007-1-216048).

[Prud'hommeaux and Seaborne, 2008] Prud'hommeaux, E. and Seaborne, A. (2008). SPARQL query language for RDF. Recommendation, W3C. `http://www.w3.org/TR/rdf-sparql-query/`.

[Solero et al., 2010] Solero, J. F. G., Ureña, M. R., Ortega, S., Lopez, J., Urmetzer, F., Hoyer, V., and Möller, K. (2010). FAST requirements specification. Deliverable D2.3.2, FAST Project (FP7-ICT-2007-1-216048).

[Taivalsaari, 1997] Taivalsaari, A. (1997). Classes vs. prototypes — some philosophical and historical observations. *Journal of Object-Oriented Programming*, 10(7):44–50.

[Tummarello et al., 2007] Tummarello, G., Morbidoni, C., Bachmann-Gmür, R., and Erling, O. (2007). RDFSync: Efficient remote synchronization of RDF models. In *6th International Semantic Web Conference and 2nd Asian Semantic Web Conference (ISWC+ASWC2007), Busan, South Korea*, pages 537–551.

[Ureña and Solero, 2010] Ureña, M. R. and Solero, J. F. G. (2010). FAST complex gadget architecture. Deliverable D3.1.2, FAST Project (FP7-ICT-2007-1-216048).

[Urmetzer et al., 2010] Urmetzer, F., Delchev, I., Hoyer, V., Janner, T., Rivera, I., Möller, K., Aschenbrenner, N., Fradinho, M., and Lizcano, D. (2010). State of the art in gadgets, semantics, visual design, SWS and catalogs. Deliverable D2.1.2, FAST Project (FP7-ICT-2007-1-216048).

[Uschold and Gruninger, 1996] Uschold, M. and Gruninger, M. (1996). ONTOLOGIES: Principles, methods and applications. *Knowl. Eng. Rev.*, 11(2).

[Wang et al., 2006] Wang, T., Parsia, B., and Hendler, J. (2006). A survey of the Web ontology landscape. In *5th International Semantic Web Conference (ISWC2006), Athens, GA, USA*, pages 682–694. Springer.

[Wolf and Wicksteed, 1997] Wolf, M. and Wicksteed, C. (1997). Date and time formats. Note, W3C. Regarding ISO 8601. `http://www.w3.org/TR/NOTE-datetime`.