FAST AND ADVANCED STORYBOARD TOOLS

FP7-ICT-2007-1-216048

http://fast.morfeo-project.eu

Deliverable D2.2.1

Ontology and Conceptual Model for the Semantic Characterisation of Complex Gadgets

Knud Möller, NUIG
Ismael Rivera, NUIG

Date: 27/02/2009

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 ready for external review

Executive Summary

This deliverable defines the ontology and conceptual model for the semantic characterisation of complex gadgets in the FAST project. As such, it feeds directly into the various implementation efforts within the project. In the document, we present the ontology development methodology (Methontology) and design principles we have applied. Following Methontology, our design and development process results in a number of different living documents, which both function 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.

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**	D2.2.1	**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**	28/02/2009	**Submitted**	27/02/2009
Status			final	
Dissemination Level	Public ☒ / Consortium ☐			
Short description of contents	D2.2.1 is the first 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 document comprises a discussion of the ontology development methodology used, a number of intermediate representations which document the development process, and the final product in the form of an OWL ontology.			
Authors	Knud Möller, NUIG, Ismael Rivera, NUIG			
Deliverable Owner	Knud Möller, NUIG		**email**	knud.moeller@deri.org
(Partner)			**phone**	+353 91 495086
Keywords	FAST, conceptual model, ontology, OWL, RDFS			

Table of contents

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.

While following an ontology design methodology which is agnostic to particular data models or implementation languages (see below), 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 also refer to what *widget* at the same time.

1.2 Structure of the Document

This deliverable is structured as follows: as part of the introduction, we will present the methodology we chose to adopt for the development of the FAST gadget ontology, by discussing each of its proposed stages in detail. Following, we will conclude the introduction by elaborating more on some of the general design decisions we are following. 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., 2009] — the structure of the remainder of this document follows directly from our adopted 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 ontology in its final form (at this moment of the project). Finishing the deliverable, we present our conclusions and outlook on years two and three of the project in Sect. 5.

1.3 *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, and a looseness in its requirements on how strictly one needs to follow the individual phases and steps. 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 acquisistion*, *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.

1.3.1 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 regard-

ing 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 [Villoslada, 2009]!).

1.3.2 Knowledge Acquisition

Knowledge Acquisition in the context of Methontology is a loose collective term to 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.

1.3.3 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 de-

tailed 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.

1.3.4 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 a 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.

1.3.5 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*).

1.3.6 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 work carried out in WP6 (Experimentation and Evaluation) in years two and three of the project.

1.3.7 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 is at the same time the ontology itself as well as its documentation.

1.4 General Design Decisions

1.4.1 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.

1.4.2 Modularisation

Apart from reusing existing terms, another basic design principle which helps to improve accessibility and understanding of the ontology is modularisation. While all terms will (for the time being) reside in the same namespace, classes and properties will nevertheless be grouped according to the specific sub-task which they belong to. E.g., terms will be grouped into defining *resources*, *pre- and post-conditions*, *general annotation* or *users and user profiles*.

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 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., 2009], and we refer the reader to this deliverable for more detail.

3 Domain Analysis

In this section, we will apply a number of the phases proposed in Methontology 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 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. Form 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*, *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 [Solero and Ortega, 2009], FAST Requirements Specification [Villoslada, 2009], FAST State-of-the-art Document [Urmetzer et al., 2009], EzWeb documentation [Lizcano et al., 2008]

3.2 Conceptualisation

In the conceptualisation phase, we first produce a *glossary of terms* which lists all classes and properties of our ontology. The glossary is seeded from the specification document (see previous section), but groups the terms according to relatedness. Also, new terms are added to this document, whereas the specification document stays the same.

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 this term. For example, while *label* is a type of thing that is important in FAST, it will not necessarily be represented as an `rdfs:Class` in the implementation step. It could just as likely be represented as a property.

Components/Resources

- `Component` (or `Resource`) - Anything that is part of a gadget (or the gadget itself). Tentatively anything that can be "touched" and moved around in the FAST IDE.
  - `Screen_Flow` - A set of screens from which a gadget for a given target platform can be generated.
  - `Screen` - An individual screen.
  - `Backend_Service` - A Web service which provides data and/or functionality to a screen. A backend service will often be external to FAST, and will probably have to be wrapped by the screen.
  - `Flow_Control_Element` - Any kind of component which can restrict the default flow of screens in a gadget.
    - `Screen_Flow_Start` - The entry point to a wigdet; the first screen.
    - `Screen_Flow_End` - A screen that ends the workflow of the gadget.
    - `Connector` - An explicit connection between two screens.
  - `Operator` - Any kind of component that is used to connect backend services to form elements. Examples are simple pipes, aggregators or various kinds of filters. Operators

concern the inner mechanics of a screen and will be covered in years 2 and 3 of the project.

  - `Form_Element` - Form elements are UI elements in a particular screen.

Pre- and Post-conditions

- `Condition` - The pre- or post-condition of either a screen or a screenflow. In the latter case, 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*.
- `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" [Solero and Ortega, 2009] In terms of RDF, a fact will probably be one statement consisting of *(S, P, O)*.

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.
- `Image` - Images are any kind of graphical object, such as icons, screenshots, etc.

Users

- `User` - A human user of the FAST IDE.
- `User_Profile` - The settings and data known about a particular user.

Properties All properties are listed in Tab. 2 (many of the properties covered in this table will also have an inverse).

Table 2: FAST Glossary of Terms, Properties

Name	Description
Components/Resources	
`contains`	Many kinds of components in FAST can contain other components: screenflows contain screens, screens contain forms or form elements, etc.

Name	Description
hasPreCondition	This property links a screen or screenflow 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.
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.

Name	Description
`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.

3.3 Integration

In this section we will investigate out 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 Friend of a Friend (FOAF) ontology, Semantically Interlinked Online Communities (SIOC) and the Dublin Core metadata specification. The outcome of this process will be a living *integration document* at 3.3.4.

3.3.1 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 1 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 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

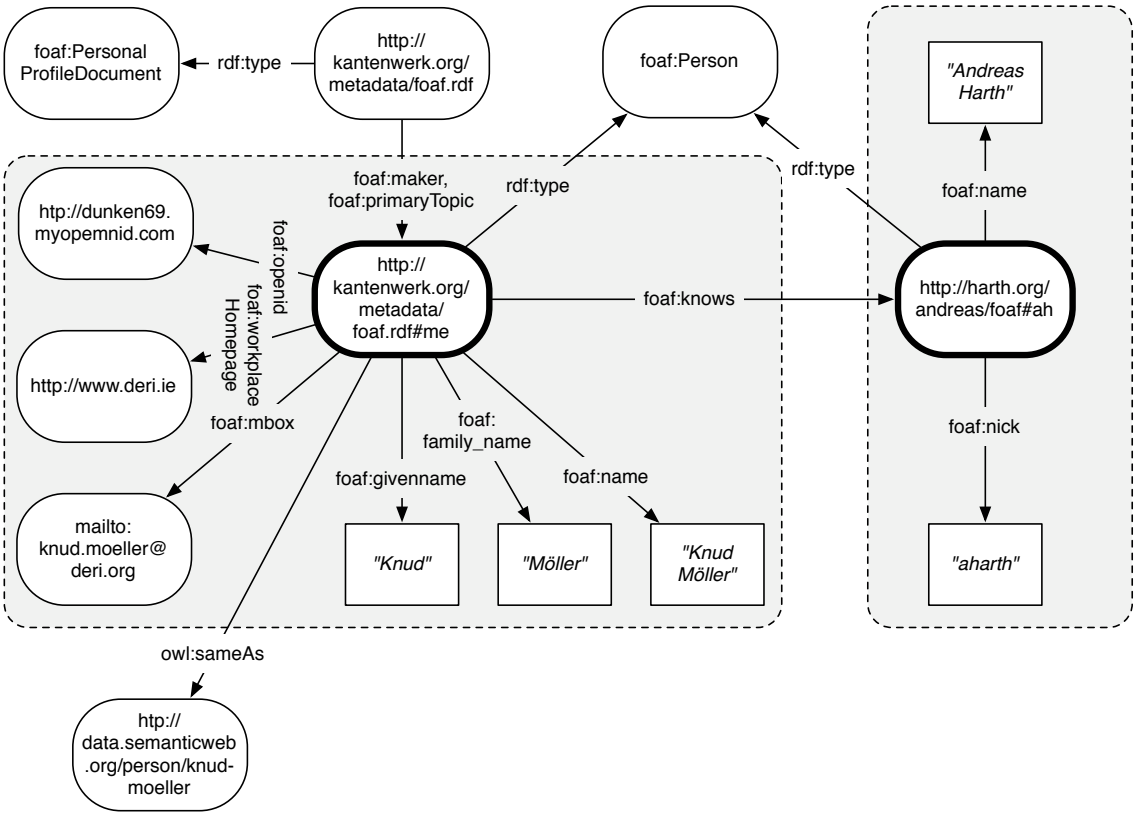

Figure 1: Example of FOAF data

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 `fast:hasIcon` is a specialisation of `foaf:depiction`. In terms of RDFS, we will say that `<fast:hasIcon> <rdfs:subPropertyOf> <foaf:depiction>`.

3.3.2 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. 2.

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 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

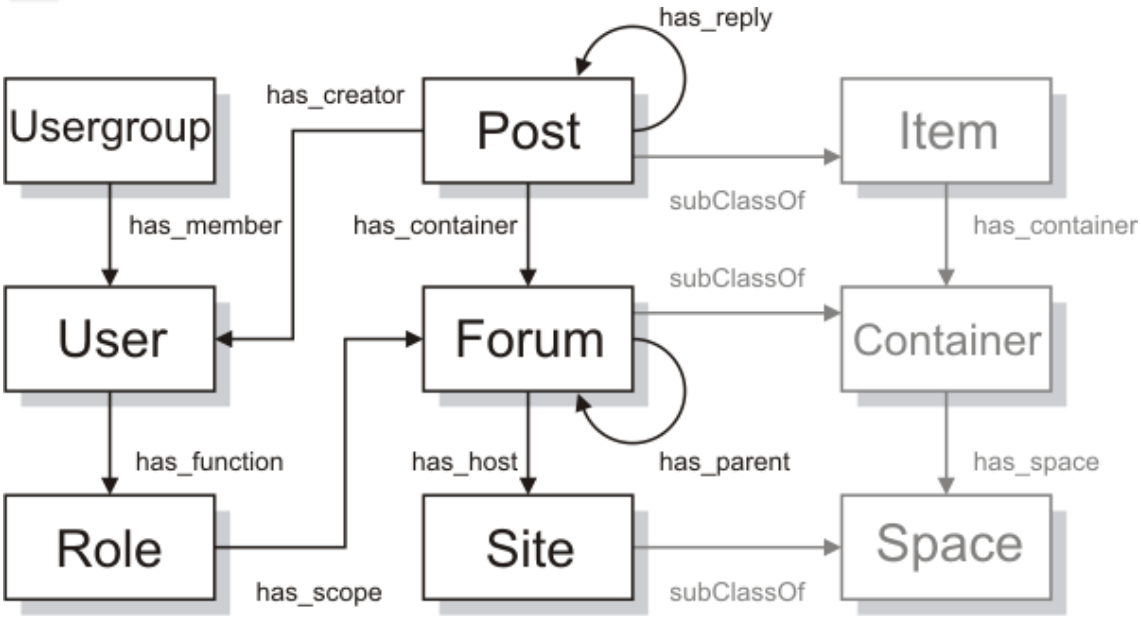

Figure 2: Overview of the SIOC ontology

document. These additional requirements could then be fulfilled by the integration of further terms from SIOC.

3.3.3 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. 3. 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).

3.3.4 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 3: 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	—
		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. 3).
hasDescription	DC	dcterms:description	—
hasCreationDate	DC	created	Dates should be formatted according to ISO 8601.
hasRights	DC	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.
hasDomainContext	DC	subject	DC specifies that subject should be non-literals. To allow simple tagging, we should represent tags as resources (TODO).
hasImage	FOAF	depiction	The inverse foaf:depicts could also be used.
hasName	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	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	interest	—
hasUserName	FOAF	accountName	—

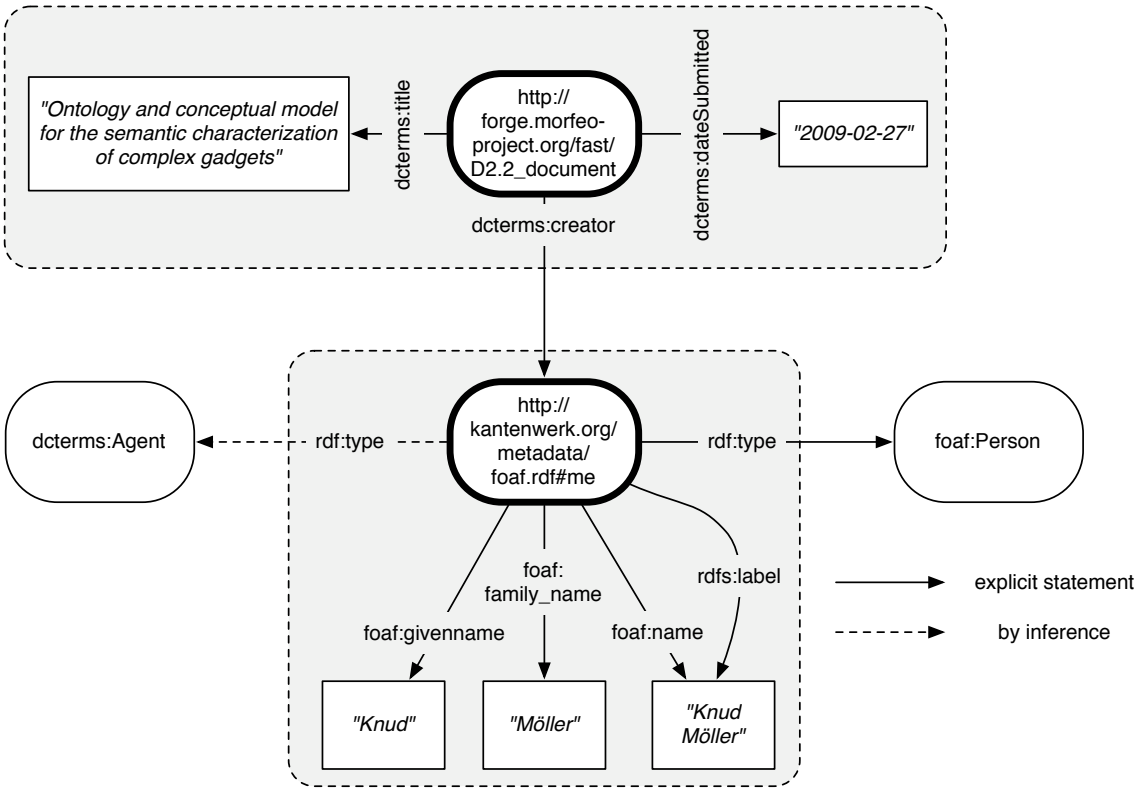

Figure 3: Example of Dublin Core data

4 The FAST Gadget Ontology

In the following sections we will define the FAST gadget ontology.

4.1 Defining Pre- and Post-conditions

The facts which define the pre- and post-conditions of screens and screenflows in FAST will be modelled as individual RDF triples. In effect, this means that conditions, which are sets of facts, can be modelled as graph patterns using SPARQL notation (see [Prud'hommeaux and Seaborne, 2008]). E.g., a simple pre-condition such as *"There has to be a user"* could be expressed as in example 1. 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"*. For the pre-condition to be fulfilled, it needs to be executed against the RDF graph in question (most likely the facts on the current canvas).

***Example* 1**

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

Post-conditions will be expressed in a similar fashion. At design time, post-condition patterns will have variable in the same way as pre-condition patterns. When a screen is added 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 will be 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 example 2, 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."*).

***Example* 2**

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

Once added to the canvas in the GVS ("instantiated at design time"), each variable would be replaced with a blank node identifier, as shown in example 3. The pre-condition in example 1 could now be executed successfully as a SPARQL query against the canvas graph.

***Example* 3**

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

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 in a graph such as in example 4.

***Example* 4**

```
<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 Möller" .
```

Since graph patterns with variable cannot directly be expressed in RDF (short of using blank nodes, which cannot have labels), the graph patterns are expressed as RDF literals and linked `fast:Condition` instances using the `fast:hasPattern` property. Each such condition will be linked to a screen or screenflow using either the `fast:hasPreCondition` or `fast:hasPostCondition` property.

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 `fast: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. 4.

Table 4: Different semantics for `isPositive`

	pre-condition	post-condition
isPositive "YES"	condition must be fulfilled	condition instantiated
isPositive "NO"	¬condition must be fulfilled	instantiated condition removed

4.2 Classes

This section contains a definition of all classes defined in the `fast` namespace. For classes integrated from other ontologies, see the integration document 3.3.4.

4.2.1 Class: `fgo:Resource`

label	Resource
description	Anything that is part of a gadget (or the gadget itself). Tentatively anything that can be 'touched' and moved around in the FAST IDE.
super_class_of	`fgo:ScreenFlow`, `fgo:Screen`, `fgo:FlowControlElement`, `fgo:ScreenComponent`, `fgo:Condition`, `fgo:Fact`, `fgo:WithPreConditions`, `fgo:WithPostConditions`, `fgo:WithConditions`, `fgo:Precondition`, `fgo:Postcondition`, `fgo:Pipe`, `fgo:Trigger`
in_domain_of	`fgo:contains`, `fgo:hasIcon`, `fgo:hasScreenshot`, `fgo:hasTag`, `fgo:hasVersion`, `fgo:hasId`, `fgo:hasName`, `fgo:hasType`, `fgo:hasDefinition`

4.2.2 Class: `fgo:ScreenFlow`

label	Screen Flow
description	The complete gadget, a set of screens.
sub_class_of	`fgo:Resource`

4.2.3 Class: `fgo:Screen`

label	Screen
description	An individual screen.
sub_class_of	`fgo:Resource`
in_domain_of	`fgo:hasCode`

4.2.4 Class: `fgo:FlowControlElement`

label	Flow Control Element
description	Any kind of component which can restrict the default flow of screens in a gadget.
super_class_of	`fgo:ScreenFlowStart`, `fgo:ScreenFlowEnd`, `fgo:Connector`
sub_class_of	`fgo:Resource`

4.2.5 Class: `fgo:ScreenFlowStart`

label	Screen Flow Start
description	The entry point to a wigdet; the first screen.
sub_class_of	`fgo:FlowControlElement`

4.2.6 Class: fgo:ScreenFlowEnd

label	Screen Flow End
description	A screen that ends the workflow of the gadget.
sub_class_of	fgo:FlowControlElement

4.2.7 Class: fgo:Connector

label	Connector
description	An explicit connection between two screens.
sub_class_of	fgo:FlowControlElement

4.2.8 Class: fgo:ScreenComponent

label	Screen Component
description	A screen component is any resource which is part of a particular screen.
super_class_of	fgo:Operator, fgo:FormElement, fgo:BackendService
sub_class_of	fgo:Resource
in_domain_of	fgo:hasAction, fgo:hasTrigger, fgo:hasLibrary

4.2.9 Class: fgo:Operator

label	Operator
description	Any kind of component that is used to connect backend services to form elements. Examples are simple pipes, aggregators or various kinds of filters.
sub_class_of	fgo:ScreenComponent

4.2.10 Class: fgo:FormElement

label	Form Element
description	Form elements are UI elements in a particular screen.
sub_class_of	fgo:ScreenComponent

4.2.11 Class: fgo:BackendService

label	Backend Service
description	A Web service which provides data and/or functionality to a screen. A backend service will often be external to FAST, and will probably have to be wrapped by the screen.
sub_class_of	fgo:ScreenComponent

4.2.12 Class: fgo:Condition

label	Condition
description	The pre- or post-condition of either a screen or a screenflow. In the latter case, 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 can be seen as a RDF bag of facts, where every fact has to be true for the condition be true as well.
sub_class_of	fgo:Resource
in_range_of	fgo:hasPreCondition, fgo:hasPostCondition, fgo:hasCondition

4.2.13 Class: fgo:Fact

label	Fact
description	A fact is the atomic formal representation of a part of a condition. Therefore, several facts compose a condition.
sub_class_of	fgo:Resource
in_domain_of	fgo:hasPattern, fgo:hasPatternString, fgo:isPositive

4.2.14 Class: fgo:WithPreConditions

label	With-precondition
description	Those kinds of resource which can have pre-conditions (i.e., screens and screen flows).
sub_class_of	fgo:Resource
in_domain_of	fgo:hasPreCondition
unionOf	fgo:ScreenFlow, fgo:Screen, fgo:Action

4.2.15 Class: fgo:WithPostConditions

label	With-precondition
description	Those kinds of resource which can have pre-conditions (i.e., screens and screen flows).
sub_class_of	fgo:Resource
in_domain_of	fgo:hasPostCondition
unionOf	fgo:ScreenFlow, fgo:Screen, fgo:ScreenComponent

4.2.16 Class: fgo:WithConditions

label	With-condition
description	Those kinds of resource which can have both pre- or post-conditions.
sub_class_of	fgo:Resource
unionOf	fgo:WithPreConditions, fgo:WithPostConditions

4.2.17 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 (i.e., the action 'showTable' will be performed when data from a service is received).
in_domain_of	fgo:hasUse
in_range_of	fgo:hasAction

4.2.18 Class: fgo:Library

label	Action
description	An action represents a specific routine which will be performed when a certain condition is fulfilled within a certain screen component (i.e., the action 'showTable' will be performed when data from a service is received).
in_domain_of	fgo:hasLanguage, fgo:hasSource
in_range_of	fgo:hasLibrary

4.2.19 Class: fgo:Precondition

label	Precondition
description	A precondition is a satisfied condition within a screenflow. It can be seen as an input of the screenflow.
sub_class_of	fgo:Resource
in_domain_of	fgo:hasCondition

4.2.20 Class: fgo:Postcondition

label	Postcondition
description	A postcondition is a result condition within a screenflow. It can be seen as an output of the screenflow.
sub_class_of	fgo:Resource
in_domain_of	fgo:hasCondition

4.2.21 Class: fgo:Definition

label	Resource definition
description	Structural and behaviour definition of a resource.
super_class_of	fgo:ScreenDefinition
in_domain_of	fgo:contains
in_range_of	fgo:hasDefinition

4.2.22 Class: fgo:ResourceReference

label	Resource reference
description	It's a reference to a certain resource. Needs a 'id' for internal identification for the resource which is referencing it, and the 'uri' of the resource.
in_domain_of	fgo:hasId, fgo:hasUri
in_range_of	fgo:contains, fgo:hasUse

4.2.23 Class: fgo:ScreenDefinition

label	Screen definition
description	Behaviour definition of a screen. This will contain which form, operators and backend services the screen is composed, and how they are connected.
sub_class_of	fgo:Definition

4.2.24 Class: fgo:Pipe

label	pipe or connector
description	Define a pipe or connector between two resources. The connection is made specifying the conditions which will be connected.
sub_class_of	fgo:Resource
in_domain_of	fgo:hasIdBBFrom, fgo:hasIdConditionFrom, fgo:hasIdBBTo, fgo:hasIdConditionTo, fgo:hasIdActionTo

4.2.25 Class: fgo:Trigger

label	Trigger
description	Define a...
sub_class_of	fgo:Resource
in_domain_of	fgo:hasIdBBFrom, fgo:hasIdBBTo, fgo:hasIdActionTo, fgo:hasNameFrom

4.2.26 Class: fgo:FormDefinition

label	Form Definition

4.2.27 Class: fgo:OperatorDefinition

label	Operator Definition

4.2.28 Class: fgo:BackendServiceDefinition

label	Backend Service Definition

4.3 Properties

This section contains a definition of all properties defined in the fast namespace. For properties integrated from other ontologies, see the integration document 3.3.4.

4.3.1 Property: `fgo:contains`

label	contains
description	Many kinds of components in FAST can contain other components: screenflows contain screens, screens contain forms or form elements, etc. It points to a resource reference to give a unique 'id' to each component within the resource.
type	`owl:ObjectProperty`
domain	`fgo:Resource`, `fgo:Definition`
range	`fgo:ResourceReference`

4.3.2 Property: `fgo:hasPreCondition`

label	has pre-condition
description	This property links certain type of resources to its pre-condition, i.e., the facts that need to be fulfilled in order for this screen or screenflow to be reachable.
type	`owl:ObjectProperty`
domain	`fgo:WithPreConditions`
range	`fgo:Condition`, `rdf:Bag`

4.3.3 Property: `fgo:hasPostCondition`

label	has post-condition
description	This property links certain type of resources to its post-condition, i.e., the facts that are produced once the screen or screenflow has been executed.
type	`owl:ObjectProperty`
domain	`fgo:WithPostConditions`
range	`fgo:Condition`, `rdf:Bag`

4.3.4 Property: `fgo:hasPattern`

label	has pattern
description	This property links a screen or screenflow to its pre-condition, i.e., the facts that need to be fulfilled in order for this screen or screenflow to be reachable.
type	`owl:ObjectProperty`
domain	`fgo:Fact`
range	`rdfs:Resource`

4.3.5 Property: fgo:hasIcon

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

4.3.6 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:Resource
range	foaf:Image

4.3.7 Property: fgo:hasCode

label	has code
description	URL of the executable code of a particular screen.
type	owl:ObjectProperty
domain	fgo:Screen
range	foaf:Document

4.3.8 Property: fgo:hasCondition

label	has condition
description	This property links a precondition or postcondition to a certain condition.
type	owl:ObjectProperty
domain	fgo:Precondition, fgo:Postcondition
range	fgo:Condition, rdf:Bag

4.3.9 Property: `fgo:hasAction`

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

4.3.10 Property: `fgo:hasUse`

label	has use string
description	This property indicates concepts used within a building block, without being a pre/postcondition.
type	`owl:ObjectProperty`
domain	`fgo:Action`
range	`fgo:ResourceReference`

4.3.11 Property: `fgo:hasLibrary`

label	has library
description	This property indicates which libraries are used by a screen component.
type	`owl:ObjectProperty`
domain	`fgo:ScreenComponent`
range	`fgo:Library`

4.3.12 Property: `fgo:hasSource`

label	has source
description	URL of the source code of a particular library.
type	`owl:ObjectProperty`
domain	`fgo:Library`
range	`foaf:Document`

4.3.13 Property: fgo:hasUri

label	has uri
description	URI of a particular resource pointing from a reference within another resource.
type	owl:ObjectProperty
domain	fgo:ResourceReference
range	foaf:Document

4.3.14 Property: fgo:hasDefinition

label	has definition
description	The structure and behaviour of a resource.
type	owl:ObjectProperty
domain	fgo:Resource
range	fgo:Definition

4.3.15 Property: fgo:integratesTerm

label	integratesTerm
description	A way to explicitly say that an ontology uses terms from another namespace. Maybe a bit redundant, but why not.
type	owl:ObjectProperty

4.3.16 Property: fgo:hasPatternString

label	has pattern string
description	This property is the textual representation of a condition.
type	owl:DatatypeProperty
domain	fgo:Fact
range	xsd:string

4.3.17 Property: fgo:isPositive

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

4.3.18 Property: fgo:hasTag

label	has tag
description	A tag is used to annotate keywords of a particular resource.
type	owl:DatatypeProperty
domain	fgo:Resource
range	xsd:string

4.3.19 Property: fgo:hasVersion

label	has version
description	The version of a particular screen or screenflow, such as 1.0, 0.2beta, etc.
type	owl:DatatypeProperty
domain	fgo:Resource
range	xsd:string

4.3.20 Property: fgo:hasTrigger

label	has trigger string
description	This property represents the events fired within a building block. e.g. a button to clear the shopping cart will fired a "clear cart" trigger.
type	owl:DatatypeProperty
domain	fgo:ScreenComponent
range	xsd:string

4.3.21 Property: `fgo:hasLanguage`

label	has language string
description	This property is the language the library is written in.
type	`owl:DatatypeProperty`
domain	`fgo:Library`
range	`xsd:string`

4.3.22 Property: `fgo:hasId`

label	has id
description	The internal ID of the resource within the container resource.
type	`owl:DatatypeProperty`
domain	`fgo:Resource`, `fgo:ResourceReference`
range	`xsd:string`

4.3.23 Property: `fgo:hasName`

label	has name
description	The name the user gives to the resource.
type	`owl:DatatypeProperty`
domain	`fgo:Resource`
range	`xsd:string`

4.3.24 Property: `fgo:hasType`

label	has type
description	The type of the resource.
type	`owl:DatatypeProperty`
domain	`fgo:Resource`
range	`xsd:string`

4.3.25 Property: `fgo:hasIdBBFrom`

label	has id building block from
description	building block id of the ‘from’ point of the pipe or trigger.
type	`owl:DatatypeProperty`
domain	`fgo:Pipe`, `fgo:Trigger`
range	`xsd:string`

4.3.26 Property: `fgo:hasIdConditionFrom`

label	has id condition from
description	condition id of the ‘from’ point of the pipe.
type	`owl:DatatypeProperty`
domain	`fgo:Pipe`
range	`xsd:string`

4.3.27 Property: `fgo:hasIdBBTo`

label	has id building block to
description	building block id of the ‘to’ point of the pipe or trigger.
type	`owl:DatatypeProperty`
domain	`fgo:Pipe`, `fgo:Trigger`
range	`xsd:string`

4.3.28 Property: `fgo:hasIdConditionTo`

label	has id condition to
description	condition id of the ‘to’ point of the pipe.
type	`owl:DatatypeProperty`
domain	`fgo:Pipe`
range	`xsd:string`

4.3.29 Property: fgo:hasIdActionTo

label	has id action to
description	action id of the 'to' point of the pipe or trigger.
type	owl:DatatypeProperty
domain	fgo:Pipe, fgo:Trigger
range	xsd:string

4.3.30 Property: fgo:hasNameFrom

label	has name from
description	name of the trigger.
type	owl:DatatypeProperty
domain	fgo:Trigger
range	xsd:string

4.4 Extension towards Specific Domains

At this point in the development of 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 resource), 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 defined in the Integration Document in Sect. 3.3.4, domain contexts are given using the *dcterms:subject* property. One option for finding domain identifiers to be used with this property is the use of DBpedia [Auer et al., 2007] 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 could also be used). As an alternative or supplement to DBpedia identifiers, it is possible to use structured tagging approaches such as MOAT[1].

[1] http://moat-project.org/

- *User Interests:* In the same way as domain contexts, a user's interests can be specified using DBpedia identifiers or structured tagging. Both can then be used in combination to suggest 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 and 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, 2009].

5 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 well-known methodology for ontology development. During the course of this process, a number of living documents — the requirements specification document, the glossary of terms, the integration document and finally the implementation document — were produced. Those documents serve as a tool for development and as documentation at the same time, and will be extended and updated in the coming two years of the project and beyond. The driving force behind these changes will be testing and evaluation of the ontology within the FAST project, and possibly in other contexts as well.

A solution for representing pre- and post-conditions of screens and screen flows — a central

aspect of FAST technology — has been proposed, which will now have to be implemented and tested both within the FAST catalogue and the FAST GVS.

Finally, to contribute to a lasting legacy of the FAST project, the gadget ontology will serve as input to standardisation efforts for the definition of semantic gadgets, which will go beyond the scope of the project itself.

References

[Ambrus, 2009] Ambrus, O. (2009). Mediation amongst ontologies: Application to the FAST ontology. Deliverable D2.4.1, FAST Project (FP7-ICT-2007-1-216048).

[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.

[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`.

[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.

[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 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.

[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 and Ortega, 2009] Solero, J. F. G. and Ortega, S. (2009). FAST complex gadget architecture. Deliverable D3.1.1, FAST Project (FP7-ICT-2007-1-216048).

[Urmetzer et al., 2009] Urmetzer, F., Delchev, I., Hoyer, V., Janner, T., Rivera, I., Aschenbrenner, N., Fradinho, M., and Lizcano, D. (2009). State of the art in gadgets, semantics, visual design, SWS and catalogs. Deliverable D2.1.1, 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).

[Villoslada, 2009] Villoslada, E. (2009). FAST requirements specification. Deliverable D2.3.1, FAST Project (FP7-ICT-2007-1-216048).

Appendix A (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 Möller, 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: fgo2009-11-16.ttl 387 2009-12-11 18:56:10Z 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/> .

<http://purl.oclc.org/fast/ontology/gadget> a owl:Ontology ;
    rdfs:label "The FAST Gagdet Ontology v0.1"@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 ;
    dcterms:created "2009-02-09"@en ;
    dcterms:modified "$Date: 2009-12-11 18:56:10 +0000 (Fri, 11 Dec 2009) $"@en ;
    dcterms:creator <http://kantenwerk.org/metadata/foaf.rdf#me> ;
    dcterms:contributor fgo:Ismael ;
    dcterms:contributor fgo:fast_members ;
    foaf:maker <http://kantenwerk.org/metadata/foaf.rdf#me> ;
    doap:revision "$Revision: 387 $" .

<http://kantenwerk.org/metadata/foaf.rdf#me> a foaf:Person ;
    foaf:name "Knud Möller"@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> .

# Classes

fgo:Resource a owl:Class ;
rdfs:label "Resource"@en ;
rdfs:comment """Anything that is part of a gadget (or the gadget itself). Tentatively
    anything that can be 'touched' and moved around in the FAST IDE."""@en .

fgo:ScreenFlow a owl:Class ;
rdfs:subClassOf fgo:Resource ;
rdfs:label "Screen Flow"@en ;
rdfs:comment "The complete gadget, a set of screens."@en .

fgo:Screen a owl:Class ;
rdfs:subClassOf fgo:Resource ;
rdfs:label "Screen"@en ;
rdfs:comment "An individual screen."@en .

fgo:FlowControlElement a owl:Class ;
rdfs:subClassOf fgo:Resource ;
rdfs:label "Flow Control Element"@en ;
rdfs:comment """Any kind of component which can restrict the default flow of screens
    in a gadget."""@en .

fgo:ScreenFlowStart a owl:Class ;
rdfs:subClassOf fgo:FlowControlElement ;
rdfs:label "Screen Flow Start"@en ;
rdfs:comment "The entry point to a wigdet; the first screen."@en .

fgo:ScreenFlowEnd a owl:Class ;
rdfs:subClassOf fgo:FlowControlElement ;
rdfs:label "Screen Flow End"@en ;
rdfs:comment "A screen that ends the workflow of the gadget."@en .

fgo:Connector a owl:Class ;
rdfs:subClassOf fgo:FlowControlElement ;
rdfs:label "Connector"@en ;
rdfs:comment "An explicit connection between two screens."@en .

fgo:ScreenComponent a owl:Class ;
rdfs:subClassOf fgo:Resource ;
rdfs:label "Screen Component"@en ;
rdfs:comment "A screen component is any resource which is part of a particular screen."@en .

fgo:Operator a owl:Class ;
rdfs:subClassOf fgo:ScreenComponent ;
rdfs:label "Operator"@en ;
rdfs:comment """Any kind of component that is used to connect backend services to
    form elements. Examples are simple pipes, aggregators or various kinds of
    filters."""@en .

fgo:FormElement a owl:Class ;
rdfs:subClassOf fgo:ScreenComponent ;
rdfs:label "Form Element"@en ;
rdfs:comment "Form elements are UI elements in a particular screen."@en .

fgo:BackendService a owl:Class ;
rdfs:subClassOf fgo:ScreenComponent ;
rdfs:label "Backend Service"@en ;
```

```
rdfs:comment """A Web service which provides data and/or functionality to a screen.
    A backend service will often be external to FAST, and will probably have to be
    wrapped by the screen."""@en .

fgo:Condition a owl:Class ;
rdfs:subClassOf fgo:Resource ;
rdfs:label "Condition"@en ;
rdfs:comment """The pre- or post-condition of either a screen or a screenflow. In
    the latter case, 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 can be seen as a RDF bag of facts, where every fact has to be true
for the condition be true as well."""@en .

fgo:Fact a owl:Class ;
rdfs:subClassOf fgo:Resource ;
rdfs:label "Fact"@en ;
rdfs:comment """A fact is the atomic formal representation of a part of a condition.
Therefore, several facts compose a condition."""@en .

fgo:WithPreConditions a owl:Class ;
rdfs:subClassOf fgo:Resource ;
owl:unionOf (fgo:ScreenFlow fgo:Screen fgo:Action) ;
rdfs:label "With-precondition"@en ;
rdfs:comment """Those kinds of resource which can have pre-conditions
    (i.e., screens and screen flows)."""@en .

fgo:WithPostConditions a owl:Class ;
rdfs:subClassOf fgo:Resource ;
owl:unionOf (fgo:ScreenFlow fgo:Screen fgo:ScreenComponent) ;
rdfs:label "With-precondition"@en ;
rdfs:comment """Those kinds of resource which can have pre-conditions
    (i.e., screens and screen flows)."""@en .

fgo:WithConditions a owl:Class ;
rdfs:subClassOf fgo:Resource ;
owl:unionOf (fgo:WithPreConditions fgo:WithPostConditions) ;
rdfs:label "With-condition"@en ;
rdfs:comment """Those kinds of resource which can have both pre- or post-conditions."""@en .

fgo:Action a owl:Class ;
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 (i.e., the action 'showTable' will
be performed when data from a service is received)."""@en .

fgo:Library a owl:Class ;
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 (i.e., the action 'showTable' will
be performed when data from a service is received)."""@en .

fgo:Precondition a owl:Class ;
rdfs:subClassOf fgo:Resource ;
rdfs:label "Precondition"@en ;
rdfs:comment """A precondition is a satisfied condition within a screenflow. It can be seen
    as an input of the screenflow."""@en .

fgo:Postcondition a owl:Class ;
rdfs:subClassOf fgo:Resource ;
```

```
rdfs:label "Postcondition"@en ;
rdfs:comment """A postcondition is a result condition within a screenflow. It can be seen
    as an output of the screenflow."""@en .

fgo:Definition a owl:Class ;
rdfs:label "Resource definition"@en ;
rdfs:comment """Structural and behaviour definition of a resource."""@en .

fgo:ResourceReference a owl:Class ;
rdfs:label "Resource reference"@en ;
rdfs:comment """It's a reference to a certain resource. Needs a 'id' for internal
identification for the resource which is referencing it, and the 'uri' of the
resource."""@en .

fgo:ScreenDefinition a owl:Class ;
rdfs:subClassOf fgo:Definition;
rdfs:label "Screen definition"@en ;
rdfs:comment """Behaviour definition of a screen. This will contain which form, operators and
backend services the screen is composed, and how they are connected."""@en .

fgo:Pipe a owl:Class ;
rdfs:subClassOf fgo:Resource ;
rdfs:label "pipe or connector"@en ;
rdfs:comment """Define a pipe or connector between two resources. The connection is made
specifying the conditions which will be connected."""@en .

fgo:Trigger a owl:Class ;
rdfs:subClassOf fgo:Resource ;
rdfs:label "Trigger"@en ;
rdfs:comment """Define a..."""@en .

### future definition of other resources ###
fgo:FormDefinition a owl:Class ;
rdfs:label "Form Definition"@en .

fgo:OperatorDefinition a owl:Class ;
rdfs:label "Operator Definition"@en .

fgo:BackendServiceDefinition a owl:Class ;
rdfs:label "Backend Service Definition"@en .


# 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 or form elements, etc.
    It points to a resource reference to give a unique 'id' to each component
    within the resource."""@en ;
rdfs:domain fgo:Resource, fgo:Definition ;
rdfs:range fgo:ResourceReference .

fgo:hasPreCondition a owl:ObjectProperty ;
rdfs:label "has pre-condition"@en ;
rdfs:comment """This property links certain type of resources to its pre-condition, i.e.,
    the facts that need to be fulfilled in order for this screen or screenflow to be
    reachable."""@en ;
rdfs:domain fgo:WithPreConditions ;
rdfs:range fgo:Condition, rdf:Bag .
```

```
fgo:hasPostCondition a owl:ObjectProperty ;
rdfs:label "has post-condition"@en ;
rdfs:comment """This property links certain type of resources to its post-condition,
    i.e., the facts that are produced once the screen or screenflow has been
    executed."""@en ;
rdfs:domain fgo:WithPostConditions ;
rdfs:range fgo:Condition, rdf:Bag .

fgo:hasPattern a owl:ObjectProperty ;
rdfs:label "has pattern"@en ;
rdfs:comment """This property links a screen or screenflow to its pre-condition,
    i.e., the facts that need to be fulfilled in order for this screen or screenflow
    to be reachable."""@en ;
rdfs:domain fgo:Fact ;
rdfs:range rdfs:Resource .

fgo:hasPatternString a owl:DatatypeProperty ;
rdfs:label "has pattern string"@en ;
rdfs:comment """This property is the textual representation of a condition."""@en ;
rdfs:domain fgo:Fact ;
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 will define when they have to be taken into account by the
inference engine or reasoner."""@en ;
rdfs:domain fgo:Fact ;
rdfs:range xsd:boolean .

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:Resource ;
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:Resource ;
rdfs:range foaf:Image .

fgo:hasTag a owl:DatatypeProperty ;
rdfs:label "has tag"@en ;
rdfs:comment """A tag is used to annotate keywords of a particular resource."""@en ;
rdfs:domain fgo:Resource ;
rdfs:range xsd:string .

fgo:hasVersion a owl:DatatypeProperty ;
rdfs:label "has version"@en ;
rdfs:comment """The version of a particular screen or screenflow, such as 1.0, 0.2beta, etc."""@en ;
rdfs:domain fgo:Resource ;
rdfs:range xsd:string .

fgo:hasCode a owl:ObjectProperty ;
rdfs:label "has code"@en ;
```

```
rdfs:comment """URL of the executable code of a particular screen."""@en ;
rdfs:domain fgo:Screen ;
rdfs:range foaf:Document .

fgo:hasCondition a owl:ObjectProperty ;
rdfs:label "has condition"@en ;
rdfs:comment """This property links a precondition or postcondition to a certain condition."""@en ;
rdfs:domain fgo:Precondition, fgo:Postcondition ;
rdfs:range fgo:Condition, rdf:Bag .

fgo:hasAction a owl:ObjectProperty ;
rdfs:label "has action"@en ;
rdfs:comment """This property indicates which actions are asociated 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 represents the events fired within a building block. e.g. a
button to clear the shopping cart will fired a "clear cart" trigger."""@en ;
rdfs:domain fgo:ScreenComponent ;
rdfs:range xsd:string .

fgo:hasUse a owl:ObjectProperty ;
rdfs:label "has use string"@en ;
rdfs:comment """This property indicates concepts used within a building block, without being a
pre/postcondition."""@en ;
rdfs:domain fgo:Action ;
rdfs:range fgo:ResourceReference .

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

fgo:hasLanguage a owl:DatatypeProperty ;
rdfs:label "has language string"@en ;
rdfs:comment """This property is the language the library is written in."""@en ;
rdfs:domain fgo:Library ;
rdfs:range xsd:string .

fgo:hasSource a owl:ObjectProperty ;
rdfs:label "has source"@en ;
rdfs:comment """URL of the source code of a particular library."""@en ;
rdfs:domain fgo:Library ;
rdfs:range foaf:Document .

fgo:hasId a owl:DatatypeProperty ;
rdfs:label "has id"@en ;
rdfs:comment """The internal ID of the resource within the container resource."""@en ;
rdfs:domain fgo:Resource, fgo:ResourceReference ;
rdfs:range xsd:string .

fgo:hasName a owl:DatatypeProperty ;
rdfs:label "has name"@en ;
rdfs:comment """The name the user gives to the resource."""@en ;
rdfs:domain fgo:Resource ;
rdfs:range xsd:string .
```

```
fgo:hasType a owl:DatatypeProperty ;
rdfs:label "has type"@en ;
rdfs:comment """The type of the resource."""@en ;
rdfs:domain fgo:Resource ;
rdfs:range xsd:string .

fgo:hasUri a owl:ObjectProperty ;
rdfs:label "has uri"@en ;
rdfs:comment """URI of a particular resource pointing from a reference within
another resource."""@en ;
rdfs:domain fgo:ResourceReference ;
rdfs:range foaf:Document .

fgo:hasDefinition a owl:ObjectProperty ;
rdfs:label "has definition"@en ;
rdfs:comment """The structure and behaviour of a resource."""@en ;
rdfs:domain fgo:Resource ;
rdfs:range fgo:Definition .

fgo:hasIdBBFrom a owl:DatatypeProperty ;
rdfs:label "has id building block from"@en ;
rdfs:comment """building block id of the 'from' point of the pipe or trigger."""@en ;
rdfs:domain fgo:Pipe, fgo:Trigger ;
rdfs:range xsd:string .

fgo:hasIdConditionFrom a owl:DatatypeProperty ;
rdfs:label "has id condition from"@en ;
rdfs:comment """condition id of the 'from' point of the pipe."""@en ;
rdfs:domain fgo:Pipe ;
rdfs:range xsd:string .

fgo:hasIdBBTo a owl:DatatypeProperty ;
rdfs:label "has id building block to"@en ;
rdfs:comment """building block id of the 'to' point of the pipe or trigger."""@en ;
rdfs:domain fgo:Pipe, fgo:Trigger ;
rdfs:range xsd:string .

fgo:hasIdConditionTo a owl:DatatypeProperty ;
rdfs:label "has id condition to"@en ;
rdfs:comment """condition id of the 'to' point of the pipe."""@en ;
rdfs:domain fgo:Pipe ;
rdfs:range xsd:string .

fgo:hasIdActionTo a owl:DatatypeProperty ;
rdfs:label "has id action to"@en ;
rdfs:comment """action id of the 'to' point of the pipe or trigger."""@en ;
rdfs:domain fgo:Pipe, fgo:Trigger ;
rdfs:range xsd:string .

fgo:hasNameFrom a owl:DatatypeProperty ;
rdfs:label "has name from"@en ;
rdfs:comment """name of the trigger."""@en ;
rdfs:domain fgo:Trigger ;
rdfs:range xsd:string .

# terms used from other ontologies

fgo:integratesTerm a owl:ObjectProperty ;
rdfs:label "integratesTerm"@en ;
```

```
rdfs:comment """A way to explicitly say that an ontology uses terms from another namespace.
    Maybe a bit redundant, but why not."""@en .

<http://purl.oclc.org/fast/ontology/gadget> fgo:integratesTerm
    sioc:User, foaf:Person,foaf:Image, foaf:OnlineAccount, foaf:holdsAccount,
    foaf:depiction, foaf:name, foaf:mbox, foaf:mbox_sha1sum, foaf:interest, foaf:accountName,
    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/cgi-bin/2003/pellet

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 .

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 .

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

Appendix B (Lists of Tables and Figures)

List of Tables

List of Figures