Development of Modular Ontologies in CASL

Klaus Lüttich[1], Claudio Masolo[2], and Stefano Borgo[2]

[1] SFB/TR8, FB3 – Dept. of Computer Science, Universität Bremen, Germany
`luettich@informatik.uni-bremen.de`
[2] Laboratory for Applied Ontology, ISTC-CNR, Trento, Italy
{`masolo,borgo`}`@loa-cnr.it`

Abstract. This paper discusses the advantages of the Common Algebraic Specification Language (CASL) for the development of modular ontologies. CASL not only offers logics with a limited expressivity like description logic, but also e.g. first-order logic and modal logic. The central part of CASL is its powerful structuring mechanism, which is orthogonal to the logical formalisms. Hence the modularization applies uniformly to various logics and its extension Heterogeneous CASL (HETCASL) has even constructs for the combination of different logics. Additionally, the Heterogeneous Tool Set (HETS) is presented which enables reasoning and manipulation of CASL specifications. By presenting a detailed example ontology used for spatial knowledge representation the benefits of specification in CASL are discussed. Furthermore, a comparison with the OWL DL import mechanism is provided.

1 Introduction

In the pioneering years of applied ontology, different practitioners and research groups started developing ontological systems. The community interested in the perspective offered by this new line of research was admittedly small and the researchers were working independently and with minimal interaction. Twenty years later, we are in a period in which the now much larger and heterogeneous community shows a deep interest in relating the different efforts. Researchers are now aware of the complexity of ontology building, maintenance, and evolution and they know that the achievement of semantic interoperability, fusion and integration of different information systems is very hard, especially in collaborative, distributed, and extremely dynamic environments such as, for example, the Web. Some important research projects are focused now on the development of methods and methodologies for addressing these problems in a more systematic way, in an attempt to create a new discipline: *ontological engineering*.

Following a trend already observed in other areas (like software engineering, object-oriented programming, and enterprise production), *modularization* is considered a central aspect in ontological engineering. The decomposition and structuring of complex ontologies into simpler components, called *modules*, is seen as a necessary step in order to (*i*) simplify the understandability, scalability, maintenance and evolution of ontologies; (*ii*) allow for the reuse of modules;

and (*iii*) improve the performance of reasoning and inference engines as in the case of question-answering in very large knowledge bases.

Despite the importance of modularization, the formal techniques for individuating, composing, and reusing modules are still quite elementary. The definition of module is itself problematic. Intuitively, a module is a relatively independent and self-contained theory that encapsulates a relevant, coherent, and integral system of interrelated concepts, i.e., for a theory to be a module, conceptual relevance, integrity and independence are required. On the one hand, modules can be considered as quite specific and stable theories linked to specific subjects/domains or purposes [11]. On the other hand, modules can be considered as quite general theories that can be integrated, specialized, and used in different domains and contexts. In this case they seem close to *components* as intended in knowledge engineering [5, 1], and they have some similarities with *knowledge patterns* [2], *design patterns* [6], and *task-specific methods* as used in reusable problem-solving methods [19] in the context of procedural knowledge. This sort of dichotomy is reminiscent of the distinction between *formal* and *material* ontology introduced by Husserl: formal ontology deals with general 'laws' characterizing relations/concepts that can be applied to different domains, while material ontology considers specific kinds of entities. In this sense, *foundational* ontologies [15] deal with the formal aspects by providing very general concepts (like event, object, quality, etc.) and relations (parthood, connection, identity, constitution, integrity, etc.), while *domain* ontologies focus on the material aspects.

If we insist on the distinction between subject (or topic) vs. general (or foundational) approaches to ontology, the modules in the first approach can be seen as sorts of building blocks that are limited in coverage, but that completely describe a specific domain. They can be composed with other modules (in the same approach) in order to increase coverage. However, since different domains can involve common general notions, a sort of alignment/integration may be needed. Conversely, the modules in the other approach, because of their generality, need to be specialized to different domains. General modules are very useful when building ontologies from scratch because they provide conceptual tools for domain analysis.

In this paper, we do not try to give a definition either of module or of ontology. To guarantee the generality of our approach, in particular to include all the kinds of modules and ontologies discussed above, we assume that both ontologies and modules are just (perhaps very complex) logical theories. Note that we do not constrain our work by specific languages as happens in many approaches driven by implementation concerns (like decidability or efficiency of the system). Indeed, often domain theories are captured in a weak language (mostly in the family of description logics) and foundational theories are given in rich languages (like first-order logic, perhaps with modal operators) [10, 21]. The core of this paper is the description of a tool, called HETS, based on the language CASL [3] that allows the specification of theories in first-order logic, modal logic, or even higher-order logics. Notice that the module development/construction and the choice of the modules to integrate (which are necessarily based on ontological principles and considerations) are carefully separated from the mapping and composition of the chosen modules.[3] We concentrate on this latter issue comparing the structuring mechanisms provided by CASL (that borrows some techniques from research in logical studies [7], software engineering [8], and applicative concerns [9]) with those of OWL DL. A complementary aspect that we leave out in this paper is the description and organization of existing modules to facilitate their retrieval. Here we assume that the user has already individuated all the modules that are to be used.

2 Logical Foundations

This section reviews the language CASL, a heterogeneous extension of CASL and CASL extensions for modal logic and description logic. Comments, labels and annotations starting with a percent symbol (%) are presented in the examples, where labels and annotations are used by HETS for the analysis. A speciality is the mixfix syntax for identifiers, which includes, for example, infix symbols like $\_\_on\_\_$.

2.1 CASL

The Common Algebraic Specification Language (CASL) [3] was developed by the Common Framework Initiative (CoFI) for algebraic specification and development formed by a large number of experts from various groups. The main goal was a standardised language for the requirements and algebraic design of software. The design of CASL has been approved by the IFIP WG 1.3 "Foundations of System Specifications". Though CASL has been designed for the specification of software requirements, it is now also used for the design of ontologies in FOL and description logic [13, 14, 12].

This paper focuses on the structuring constructs of CASL; these are quite independent of the underlying logic. Still, we start with a brief overview of the underlying logic followed by the introduction of the structuring concepts. The underlying logic is often referred to as CASL *in-the-small* or *basic specification* and it provides the specification of axioms in first-order logic (FOL) with equality or in other logics (See Sect. 2.3 for CASL extensions). Such a basic specification is also called theory and a theory is divided into a signature and sentences. All symbols for predicates, total and partial functions (called operation in CASL; hence the keyword **op**) and sorts have to be declared within the signature. A declaration of predicates and functions includes a type in terms of sorts, also called profile. Sorts are interpreted as non-empty sets. In summary, CASL offers typed predicates and functions based on declared sorts where the correct application of symbols in the axioms is statically determined. An example of a basic specification of "Services located at Rooms" is presented in Fig. 1. CASL has no

[3] For information on module extraction from ontologies see [22].

syntactic or semantic distinction between TBoxes and ABoxes. Still, the axioms can be easily divided into two groups of formulas: those that have free variables and those that have no free variables, respectively.

sorts *Service*, *Room*
pred $\_\_locatedAt\_\_ : Service \times Room$
$\forall\ s : Service \bullet \exists\ r : Room \bullet s\ locatedAt\ r$ %(Services are in rooms)%

Fig. 1. Basic Specification of "Services Located at Rooms"

Orthogonal to the CASL basic specification described above are CASL's *in-the-large* structuring constructs. Throughout the rest of this section we will use the term *symbol* for identifiers of sorts and for identifiers of predicates and functions with their respective profile and the term *local environment* for declared symbols starting with an empty local environment. The *global environment* covers the names of entire specifications and views, and the manipulation of both environments is described within the rest of this section.

CASL structuring constructs allow the specification of the modularity of libraries of specifications (theories) independently of the underlying logic. A file with CASL specifications is called a library. A library consists of named specifications and downloads from other CASL libraries where the specifications are imported into the global environment of named specifications. Downloaded specifications can be renamed. All specifications introduced in this paper share the same library header presented in Fig. 2 which denotes some optional global comments by the authors and the date of the library and downloads from published CASL libraries [3].

library ROOMSANDSERVICES

%author Klaus Lüttich, Claudio Masolo, and Stefano Borgo
%date July, August 2006

from BASIC/RELATIONSANDORDERS **get**
IRREFLEXIVERELATION, ASYMMETRICRELATION, SIMILARITYRELATION

from BASIC/NUMBERS **get** RAT

Fig. 2. The Header of our Library ROOMSANDSERVICES

New specifications within a library are named with an identifier not present in the global environment. Figure 3 presents the construction of a named specification in terms of structuring concepts. The reuse of a specification is called a *reference* and it also gets the current local environment, e.g. RAT gets the empty local environment.

```
spec DistanceAlt =
     Rat
then sort  Elem
     op    distance : Elem × Elem → Rat, comm
     ∀ x, y, z : Elem
     • distance(x, y) = 0 ⇔ x = y                              %(distance_0)%
     • distance(x, y) + distance(y, z) ≥ distance(x, z)    %(triangle_inequality)%
end
```

Fig. 3. Named Structured Specification of a Distance Function

A specification can be formed via *union* and *extension* of specifications. These specifications could be any other specification formed by the constructs introduced in this section. Within a union each specification gets the same environment. If both specifications introduce a symbol with the same identifier (and same profile for predicates and operations) these symbols are identified. This is known as the 'same name, same thing' principle. An extension always gets the local environment of the previous specification part and extends it with additional symbols and/or sentences. The identification of symbols is the same as for unions. Extensions introduced with the annotation **%implies** may only consist of theorems without a signature part where all formulas are entailed by the preceding sentences (See Sect. 4 for discussion of theorem provers).

If some symbols of two specifications should be kept separate, *renaming* is used. A renaming applies to the local environment and the sentences of that local environment are translated according to the given symbol mapping. This symbol mapping can be given in terms of identifiers or with qualified symbols (qualified with kind sort, op, pred and/or profile), e.g. if the same identifier is used as sort and predicate simultaneously. Renaming allows the reuse of a specification for different sorts or predicates (Fig. 6).

Moreover, it is possible to *hide* some auxiliary symbols, e.g. if these symbols are not needed where a specification is reused. If a sort is hidden all predicates and functions using this sort in their profile are hidden as well. Symbols can be distinguished by their kind (sort, op, pred) and/or their profile. After the hiding the model class is reduced to models without the hidden symbols. Thus a hiding might not be flattened to a single basic specification. Instead of giving a list of symbols to hide, it is also possible to *reveal* those symbols that should be kept in the theory. For convenience it is further possible to simultaneously reveal and rename the symbols by giving a symbol mapping with the reveal construct.

There are three additional constructs not mentioned above for the structuring of specifications which are not used in the examples presented. Thus, they are only mentioned here for completeness: (*i*) marking a specification as *free* yields either the class of initial models (if the local environment is empty) or a free extension; (*ii*) marking a specification as *local within* another specification implicitly hides the symbols declared in the local part; (*iii*) marking a specification as *closed* gives the possibility of starting again with an empty local environment.

Additionally, named specifications can be generic over argument specifications and can have imports of other (named) specifications to be common to the parameters and the body of the specification. An example is a specification of distances where the element sort is independent of the general axioms for a distance function and the specification RAT is an import for predicates and functions on rational numbers (see Fig. 4). This generic specification yields the same theory as DISTANCEALT in Fig. 3. The advantage is the explicit parameter. The reference to a generic specification is called *instantiation* where an actual specification must be provided for each formal parameter specification (Fig. 5).

```
spec DISTANCE[sort Elem] given RAT =
     op distance : Elem × Elem → Rat, comm
     ∀ x, y, z : Elem
     • distance(x, y) = 0 ⇔ x = y                                 %(distance_0)%
     • distance(x, y) + distance(y, z) ≥ distance(x, z)   %(triangle_inequality)%
end
```

Fig. 4. Generic Specification of a Distance Function

Furthermore, it is possible to formulate *views* between specifications, where all sentences of specification SP_1 are entailed by the theory of SP_2. The model theoretic semantics of a view is $\mathbf{Mod}(SP_2) \models \mathbf{Mod}(\sigma(SP_1))$ where σ is a symbol mapping for the translation of symbols in SP_1 to symbols in SP_2. This yields proof obligations such that each sentence of theory SP_1 must be entailed by theory SP_2. Some of the obligations can be solved on a structural level, e.g. if both specifications SP_1 and SP_2 reference the same specification SP_3 all sentences of SP_3 are present in SP_1 and SP_2. All other proof obligations have to be discharged on the basic specification level with a suitable theorem prover (see Sect. 4).

2.2 CASL structuring compared to OWL DL imports

The only structuring mechanism OWL DL offers is the import of other OWL DL ontologies allowing even cyclic imports [4, 18]. Furthermore, OWL DL allows reference to symbols in other ontologies without any import with an unspecified semantics. OWL DL lacks renaming and hiding constructs, e.g. a symbol introduced in ontology A will be present in all ontologies importing A and can never be identified with a symbol of a different ontology.

In contrast, a named CASL specification can only be used after it is defined and it is not possible to define cyclic dependencies. Only symbols declared by referenced (imported) specifications or declared previously within the specification are allowed within the sentences. Renaming and hiding of symbols is provided with a precisely defined semantics [3].

2.3 Casl extensions

The orthogonal design of the Casl basic specification and structured specification allows the plug in of various other logics as basic specifications. Because of this independence of structuring and underlying logic each of the Casl extensions presented in this section uses the same structuring facilities as presented in Sect. 2.1. Some of the other logics are just extensions or restrictions of Casl, and others might be totally different, e.g. OWL DL ($\mathcal{SHOIN}(\mathbf{D})$). We now briefly introduce two extensions of Casl relevant for ontology design and engineering:

ModalCasl is a Casl extension with multi-modalities and term modalities. It allows the specification of modal systems with Kripke's possible worlds semantics. Further, it is possible to express certain forms of dynamic logic.

Casl-DL is a restriction of a Casl extension, realizing a strongly typed variant of OWL DL in Casl syntax [14]. Cardinality restrictions for the description of sorts and unary predicates are added as new formula constructs. Despite a FOL like syntax, only quantification with a limited flow of variables is allowed, such that an equivalence between Casl-DL, OWL DL and $\mathcal{SHOIN}(\mathbf{D})$ is established. In particular, only unary and binary predicates, concepts (as subsorts of the topsort *Thing*) and predefined datatypes forming a separate hierarchy from *Thing* are allowed. Casl-DL distinguishes two types of concepts: (*i*) non-empty concepts (modeled as sorts), and (*ii*) possibly empty concepts (modeled as unary predicates). Functional roles are modeled as partial functions, giving a shortcut compared to writing down an axiom, and constant functions serve as individuals.

2.4 HetCasl

HetCasl extends Casl's structuring mechanisms allowing the combination of different logics, e.g. the specification of some aspects in FOL and other aspects in higher-order logic (HOL) and the union of these theories into a joint HOL theory. HetCasl offers facilities to choose a logic for following specifications, to embed or encode a specification along a *comorphism* into a different (sub-)logic and to project a specification to a less expressive logic. Comorphisms and projections are not given in terms of Casl specifications, instead they are introduced as functions within the Heterogeneous Tool Set Hets (see Sect. 4). An embedding of Casl-DL (including cardinalities) into Casl is presented in [14].

3 Example: Rooms and Services

In this section we will illustrate the Casl mechanisms for composing theories described in the previous section by means of one non trivial example that is relevant in the context of the Collaborative Research Center "Spatial Cognition" (SFB/TR 8). In particular, it elaborates the relation of physical space represented by robots and services used by humans in a formal language.

```
%% physical description of buildings
spec RoomsMap =
        Distance[sort Room fit Elem ↦ Room]
and     {IrreflexiveRelation and  AsymmetricRelation}
        with Elem ↦ PhysicalEntity, __~__ ↦ __on__
then sorts  Room, Floor < PhysicalEntity
        ∀ x, y : PhysicalEntity
        • x ∈ Room ⇒ ¬ x ∈ Floor                  %(Room and Floor are disjoint)%
        • x on y ⇒ x ∈ Room ∧ y ∈ Floor                      %(Room on Floor)%
        ∀ x : PhysicalEntity
        • x ∈ Floor ⇒ (∃ y : Room • y on x)     %(Floors decomposition in Rooms)%
then %implies
        ∀ x : PhysicalEntity
        • x ∈ Room ⇒ ¬ (∃ y : PhysicalEntity • y on x)   %(Nothing is on Rooms)%
        • x ∈ Floor ⇒ ¬ (∃ y : PhysicalEntity • x on y)  %(Floors are on nothing)%
end
```

Fig. 5. The Specification RoomsMap

The complete theory aims at the representation of the structure of buildings. On one side, the specification RoomsMap (Fig. 5) partially describes the *physical* structure of the floors of a building quantitatively by means of a distance relation (*distance*) between *rooms*[4] (a kind of 'spatial containers'). On the other side, the specification ServicesMap (Fig. 6) qualitatively describes the *functional* structure of the buildings, i.e. the spatial links between the *services* (like library, kitchen, chemical laboratory, etc.) present in the buildings. The ontological rationale behind the distinction between rooms and services relies on the acceptance of the fact that the same service, e.g. a chemical laboratory, might be moved from one room to another one maintaining its identity, i.e., the identity criteria for services does not take into account the spatial containers where services are located.

In RoomsMap the *distance* relation between rooms is defined by the generic specification Distance (see Fig. 4) which is instantiated assigning the sort *Room* to the parameter specification. The relation *on* between rooms and floors, a sort of incidence relations, is introduced by the renaming of the united specifications IrreflexiveRelation and AsymmetricRelation coming from an external library.

In ServicesMap two relations are considered: *near* is a purely spatial relation standing for the fact that the boundaries of the spatial locations of the two services overlap, while *connected* is a more functional relation representing the fact that it is possible to directly move from one service to another one without crossing other services. In particular, *connected* implies *near* but not vice versa,

[4] Note that rooms are in general different from spatial locations, in the sense, that rooms can change in space. For example, the same room can be shortened a little bit.

i.e., in order to share a door (or, more generally, an opening) two connected services need to be spatially adjacent, but two spatially adjacent services can be separated by a wall without a door. Similarly to the case of *on* in RoomsMap, both *connected* and *near* are introduced by renaming of an external specification, in this case, the specification SimilarityRelation.

```
%% functional description of buildings
spec ServicesMap =
      %% share some wall
      SimilarityRelation
      with Elem ↦ Service, __∼__ ↦ __near__
and   %% share some wall with doorway / passage
      SimilarityRelation
      with Elem ↦ Service, __∼__ ↦ __connected__
then ∀ x, y : Service
      • x connected y ⇒ x near y                 %(connected implies near)%
end
```

Fig. 6. The Specification ServicesMap

The two specifications RoomsMap and ServicesMap can be linked in order to have a richer description of buildings that merges the physical and the functional information. It is quite intuitive to consider services as located in rooms[5], but, right now, this has not been represented. The specification Services_in_Rooms (Fig. 7) makes a step toward this merging. It extends the union of RoomsMap and ServicesMap with the characterization of the primitive relation *locatedAt* between services and rooms and the definition of a distance function between services (*SDistance*). The axioms characterizing *locatedAt* guarantee that: (*i*) only one service is located at one room; (*ii*) a room can contain only one service; and (*iii*) *near* (and, as consequence, *connected*) applies only to services located at the same floor. Note that these axioms not only characterize *locatedAt* but they also better specify both *near* and *connected.* For example, it is clarified that two services situated at different floors are not in the *near* relation even though the ceiling of one service might be spatially co-located with the floor of a another service. The new function *SDistance*, defined on the basis of *locatedAt* and *distance*, adds quantitative information on services through the quantitative information on the rooms they are located at.

Additionally, the view ServicesMap_TO_RoomsMap (Fig. 8) establishes a mapping between the function *distance* (specified in RoomsMap) and the relation *near* (specified in ServicesMap), i.e., it gives a 'reading' of *near* in terms of *distance.* The view maps services to rooms and the *near* relation to the auxiliary relation *close.* Rooms in the *close* relation are on the same floor

[5] As services can move, their location may change in time. But here we do not consider time.

```
spec Services_in_Rooms =
       RoomsMap and ServicesMap
then pred __locatedAt__ : Service × Room
   ∀ s, s1, s2 : Service; r, r1, r2 : Room
   • ∃ r : Room • s locatedAt r                        %(Services are in rooms)%
   • ∃ s : Service • s locatedAt r                 %(No rooms without services)%
   • s1 locatedAt r ∧ s2 locatedAt r ⇒ s1 = s2    %(Only one service in a room)%
   • s locatedAt r1 ∧ s locatedAt r2 ⇒ r1 = r2      %(No multi-located services)%
   ∀ s1, s2 : Service; r1, r2 : Room; f1, f2 : Floor
   • s1 locatedAt r1 ∧ s2 locatedAt r2 ∧ s1 near s2 ∧ r1 on f1 ∧ r2 on f2
     ⇒ f1 = f2
   op SDistance : Service × Service → Rat
   ∀ s1, s2 : Service; r1, r2 : Room; n : Rat
   • SDistance(s1, s2) = n
     ⇔ (∃ r1, r2: Room • s1 locatedAt r1 ∧ s2 locatedAt r2 ∧ distance(r1, r2) = n)
then %implies
   ∀ s1, s2 : Service; r1, r2 : Room; f1, f2 : Floor
   • s1 locatedAt r1 ∧ s2 locatedAt r2 ∧ s1 connected s2 ∧ r1 on f1 ∧ r2 on f2
     ⇒ f1 = f2
end
```

Fig. 7. The Specification Services_in_Rooms

and at distance zero. The definition of *close* together with the characterization of the *distance* function entails that *close* is reflexive and symmetric, which are exactly the axioms given for *near* by the reference to the specification SimilarityRelation. The proof obligations generated by the view are discharged using Hets and its connected automatic theorem provers as described in Sect. 4. Note that, in the view, services are just mapped to rooms, while in the specification Services_in_Rooms they were linked to rooms by the relation *locatedAt*. Intuitively, this direct mapping is acceptable because only one service can be located at a room, and because services cannot be multi-located. But while the specification Services_in_Rooms, allows for an explicit characterization of these assumptions, in the view, they remain implicit.

4 Hets

In this section we explain the Heterogeneous Tool Set (Hets) [16]. Hets helps in understanding the structure of the whole theory, i.e. it visualizes the links and the dependencies between the specifications as development graphs. It performs a static type checking of the symbols used in formulas and of symbol mappings. Furthermore, it infers the proof obligations of a library and offers interfaces to theorem provers. A Casl library of specifications can be transformed by Hets into various different formats, like automatically formatted LaTeX (used for the examples in this paper) and ISO-latin-1. Casl has a clear and human-oriented ISO-latin-1 input syntax.

```
%% relating two modules with predicates, no identification of sorts
view ServicesMap_TO_RoomsMap :
      {ServicesMap reveal Service, __near__} to
      {RoomsMap
       then pred  close : Room × Room
             ∀ x, y : Room
             • close(x, y)
                ⇔ distance(x, y) = 0 ∧ (∃ f : Floor • x on f ∧ y on f) %(close def)%
      }
      = Service ↦ Room, __near__ ↦ close
end
```

Fig. 8. The View ServicesMap_TO_RoomsMap

Development graphs are the main interface for users to libraries of Casl specifications. Figure 9 shows the development graph of the library RoomsAnd-Services used as an example in this paper. The bold arrows with a filled arrow head denote the dependencies and development of specifications. The number of incoming arrows distinguishes unions from extensions, i.e. a single incoming arrow denotes an extension and two or more incoming arrows denote a union. Only the named theories within the development get a named node. The unnamed nodes are intermediate steps towards the development of the whole. The theory associated with unnamed node directly above the node "ServicesMap" is constructed from the union of two renamings of "SimilarityRelation". Slim arrows with an open arrow head denote proof obligations either associated with views or "**then %implies**". The view presented in Fig. 8 is shown in the development graph as the slim arrow between two unnamed nodes in the bottom right of the figure. Such development graphs can also be written in various graph representations by Hets.

Within the graphical user interface (GUI) of Hets which displays a development graph the proof obligations are moved from the edges to the nodes (theories) for proving. The proof obligations can be passed for discharging either to the semi automatic theorem prover Isabelle [17] or to an automatic theorem prover, like Spass [24] or Vampire (provided through MathServ [25]). All theorems (including those yielded by the view) which have been presented in this paper have been proved with Vampire and some were additionally proved with Spass. The support of specialized DL reasoners like FaCT++ [23] and Pellet [20] is under development.

Translations and projections between different logics are implemented as computational functions within Hets and are accessible via the development graph GUI of Hets. Hets provides reading of OWL DL documents and transformation into Casl-DL and an embedding of Casl-DL into Casl. The approximation of Casl with Casl-DL [12], and the translation of Casl-DL in OWL DL and the writing of OWL DL documents is under development.

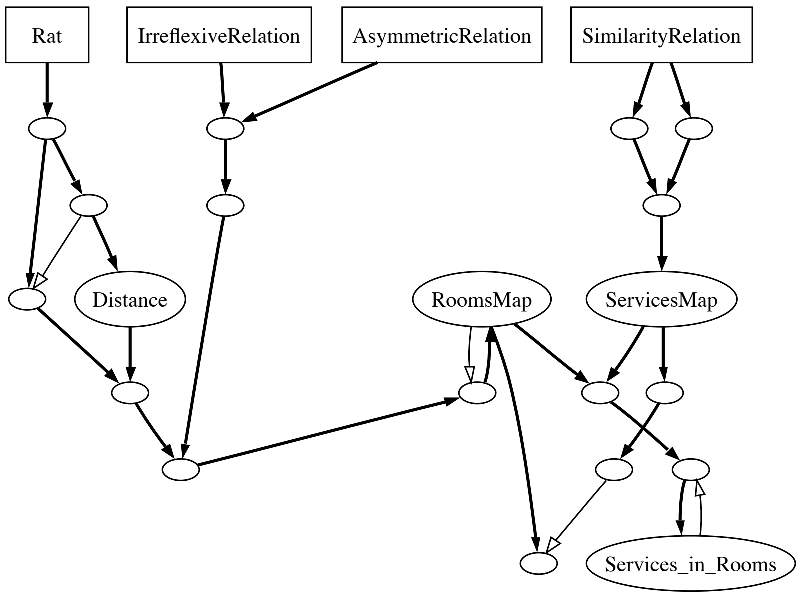

Fig. 9. Development Graph of the Library RoomsAndServices

5 User experience

The language Casl has been successfully used for the Descriptive Ontology for Linguistic and Cognitive Engineering (Dolce) [15]. While previously the developers of Dolce at the Laboratory for Applied Ontology (LOA) only used LaTeX for designing their foundational ontology Dolce, there started a collaboration with the developers of Casl and Hets in Bremen in 2004 for a redesign of Dolce as a Casl structured library. Furthermore it turned out that even some specifications of the basic Casl libraries could be reused, e.g. PartialOrder (from the library BasicRelationsAndOrders). The library Dolce is constituted of 39 specifications, where 12 smaller specifications are reused 3 or more times for the axiomatization of different concepts. The last development step is the union of 8 separate theories together with the Taxonomy specification resulting in the theory of Dolce.

Acknowledgements

This work has been supported by the DAAD-Vigoni program and the collaborative research center “Spatial Cognition” SFB/TR 8. We thank John Bateman, Nicola Guarino, Till Mossakowski, Laure Vieu, Joana Hois, and Scott Farrar for fruitful discussions.

References

1. P. Clark and B. Porter. Building concept representations from reusable components. In *National Conference on Artificial Intelligence (AAAI'97)*, Providence, Rhode Island, 1997.
2. P. Clark, J. Thompson, and B. Porter. Knowledge patterns. In A. G. Cohn, F. Giunchiglia, and B. Selman, editors, *International Conference on Principles of Knowledge Representation and Reasoning (KR'00)*, pages 591–600, Breckenridge, Colorado, USA, 2000. Morgan Kaufmann Publishers.
3. CoFI (The Common Framework Initiative). CASL *Reference Manual*. LNCS 2960 (IFIP Series). Springer Verlag; Berlin, 2004.
4. M. Dean and G. Schreiber, editors. OWL Web Ontology Language – Reference. W3C Recommendation http://www.w3.org/TR/owl-ref/, 10 February 2004.
5. B. Falkenhainer and K. Forbus. Compositional modelling: Finding the right model for the job. *Artificial Intelligence*, 51:95–143, 1991.
6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.
7. J. Garson. Modularity and relevant logic. *Notre Dame Journal of Formal Logic*, 30(2):207–223, 1989.
8. J. A. Goguen. Reusing and interconnecting software components. *IEEE Computer*, 19(2):16–28, 1986.
9. B. C. Grau, B. Parsia, E. Sirin, and A. Kalyanpur. Modularity and Web ontologies. In *International Conference on the Principles of Knowledge Represenation and Reasoning*, pages 198–209, Lake District, UK, 2006. AAAI Press.
10. N. Guarino. Formal Ontology in Information Systems. In N. Guarino, editor, *Formal Ontology in Information Systems (FOIS'98)*, pages 3–15, Trento, Italy, 1998. IOS Press.
11. M. Jarrar. *Towards Methodological Principles for Ontology Engineering*. Phd thesis, Vrije Universiteit, 2005.
12. K. Lüttich. Approximation of Ontologies in CASL. In *Formal Ontology in Information Systems (FOIS-2006)*, Frontiers in Artificial Intelligence and Applications. IOS Press; Amsterdam; http://www.iospress.nl, 2006. to appear.
13. K. Lüttich and T. Mossakowski. Specification of Ontologies in CASL. In A. C. Varzi and L. Vieu, editors, *Formal Ontology in Information Systems – Proceedings of the Third International Conference (FOIS-2004)*, volume 114 of *Frontiers in Artificial Intelligence and Applications*, pages 140–150. IOS Press; Amsterdam, 2004.
14. K. Lüttich, T. Mossakowski, and B. Krieg-Brückner. Ontologies for the Semantic Web in CASL. In J. Fiadeiro, editor, *WADT 2004*, LNCS 3423, pages 106–125. Springer Verlag; Berlin, 2005.
15. C. Masolo, S. Borgo, A. Gangemi, N. Guarino, and A. Oltramari. WonderWeb Deliverable D18: Ontology Library. Technical report, ISTC-CNR, 2003.
16. T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. Available at www.tzi.de/cofi/hets, University of Bremen.
17. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer Verlag; Berlin, 2002.
18. P. F. Patel-Schneider, P. Hayes, and I. Horrocks, editors. OWL Web Ontology Language – Semantics and Abstract Syntax. W3C Recommendation http://www.w3.org/TR/owl-semantics/, 10 February 2004.
19. G. Schreiber, H. Akkermans, A. Anjewierden, R. DeHoog, N. Shadbolt, W. Van de Velde, and B. Wielinga. *Knowledge Engineering and Management - The CommonKADS Methodology*. MIT Press, Massacusetts, 2000.

20. E. Sirin, M. Grove, B. Parsia, and R. Alford. Pellet OWL reasoner. http://www.mindswap.org/2003/pellet/index.shtml, May 2004.
21. S. Staab and R. Studer. *Handbook of Ontologies*. Springer Verlag, Berlin (DE), 2004.
22. H. Stuckenschmidt and M. Klein. Structure-based partitioning of large concept hierarchies. In A. McIlraith, Sheila, D. Plexousakis, and F. van Harmelen, editors, *International Semantic Web Conference (ISWC'04)*, pages 289–303, Hiroshima, Japan, 2004.
23. D. Tsarkov and I. Horrocks. FaCT++ description logic reasoner: System description. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 292–297. Springer, 2006.
24. C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobalt, and D. Topic. SPASS version 2.0. In A. Voronkov, editor, *Automated Deduction – CADE-18*, volume 2392 of *Lecture Notes in Computer Science*, pages 275–279. Springer Verlag; Berlin, July 27-30 2002.
25. J. Zimmer and S. Autexier. The MathServe System for Semantic Web Reasoning Services. In U. Furbach and N. Shankar, editors, *Proceedings of the third International Joint Conference on Automated Reasoning*, volume 4130 of *LNCS*, Seattle, USA, August 2006. Springer Verlag.