Modelling Episodes with Generic Ontology Design Patterns[1]

Bernd KRIEG-BRÜCKNER [a,b], Mihai CODESCU [a], Mihai POMARLAN [a]
[a] *Collaborative Research Center EASE, Universität Bremen, Germany*
[b] *German Research Center for Artificial Intelligence, Bremen, Germany*

Abstract. Developing knowledge-driven applications requires a mix of competencies; however, ontology experts are often not available, and end-users may be prone to introducing mistakes. *Generic Ontology Design Patterns*, GODPs, encapsulate complex semantics; during reuse, instantiations provide handles for checking arguments against structural and *semantic* constraints stated in ontology parameters. Development is divided according to expertise: *ontology experts* develop GODPs while *domain experts* focus on their domain application; for *end-users*, *consistency* of modelling and *safety of data input* are significantly increased.

Ontology engineering with GODPs is demonstrated with *episodes*, a significant extension of DUL patterns, for a use case in service robotics: GODPs for *narratively enabled episodic memories* provide increased consistency, and *safe population with data* substantially *scales up*.

1. Introduction

Developing knowledge-driven applications requires a mix of competencies that is not always available. In particular, not all development teams have or can afford ontology experts to handle the foundational aspects of knowledge modelling, and end-users of knowledge-driven solutions may be prone to introducing mistakes. We distinguish (at least) three kinds of stakeholders, whose kind and level of expertise is quite different: *end-users* should not be required to have ontology expertise and may have little domain knowledge; *domain experts* often lack ontology expertise due to insufficient training; *ontology experts* usually have little domain expertise.

Ontology Design Patterns (ODPs) have been proposed for some time as a methodology for ontology development, see the early work by [1,2,3], the compilation in [4,5,6], and the review of the state of the art in [7].

In theory, ODPs provide a solution for the lack of ontology experts: ODPs enable domain ontologists to reuse existing best practices and design decisions, and thus benefit from the experience of ontology experts, who developed the ODPs. However, in practice

The research reported in this paper was partially supported by the German Research Foundation (DFG), as part of the Collaborative Research Center (Sonderforschungsbereich) 1320 "EASE - Everyday Activity Science and Engineering" (`http://www.ease-crc.org/`), primarily in subproject P01: 'Embodied Semantics for the Language of Action and Change'.

the adaptation of ODPs as tools for ontology engineers has been slow. In our opinion this is caused by the fact that currently the utilisation of ODPs is cumbersome for ontology developers and not practical for large ontologies, let alone data patterns.

Generic Ontology Design Patterns, GODPs, have been proposed as a methodology for representing and instantiating ODPs in an adaptable way allowing domain experts and end-users to safely use ODPs [8,9,10,11]. In [12] we argue that GODPs implement ODPs effectively, and discuss the merits of GODPs vs. ODPs.

The main ideas behind GODPs are the following: GODPs are expressed in a dedicated formal, parameterised pattern language that allows to (a) define GODPs, (b) specify instantiations, and (c) extend and combine them to larger GODPs; they embody dedicated development operations. GODPs are defined in Generic DOL, an extension of the *Distributed Ontology, Model and Specification Language*, DOL [13], supported by the *Heterogeneous Tool Set*, Hets [14].

GODPs enable the nested use of ODPs, which reduces code duplication. Furthermore, GODP developers are able to explicitly state logical properties of GODPs, and to represent competency questions and definition extensions. During reuse, instantiations provide extra handles for improving consistency, checking arguments against structural and semantic requirements stated in the parameters.

GODPs are patterns: they contain typed variables. The definition of a GODP involves the specification of parameters that need to be provided for the instantiation of a GODP. Parameters are ontologies; the case of symbols as parameters is covered by ontologies without axioms and only one symbol declaration. These enable the expression of *powerful semantic constraints* using corresponding axioms; such requirements act like preconditions for instantiations, guaranteeing more consistency and safety. For each argument for a parameter, a verification condition is generated. If expressed in DL (as in this paper), it may be discarded automatically by deduction using a DL reasoner; in a heterogeneous setting, DOL and Hets allow more expressive logics.

Overview. In this paper we emphasise this *consistency* aspect: we argue, and show by example, that not only models expressed as GODPs may be more safely extended using e.g. subclass constraints, but also *data* expressed as individuals may be interrelated with object properties and "typed" using structural constraints, increasing safety of data input. We would like to demonstrate, how

- *ontology experts* encapsulate complex semantics in *foundational* GODPs;
- *domain experts* assemble a specialised toolbox of *configuration* GODPs;
- *end-users* focus on GODPs *dedicated to a particular development domain*.

Thus domain experts benefit from the *delegation of expertise* to ontology experts, relieved from cumbersome detail and avoiding potential mistakes due to lack of training; end-users are provided with a suitable user interface for *safe* data input.

We will guide our exposition by an illustration of how GODPs can be used in a knowledge-driven process. We have chosen a problem coming from service robotics as an example: to define activity episodes (Sect. 2). Episodes are descriptions of past events, augmented by semantic annotations about what happened. The goal of knowledge-driven development then is to define what episodes consist of, and to provide ways to ease construction of coherent episodes, regardless of whether the data for the episode come from human or autonomous robot performance. Sect. 3 briefly recalls Generic DOL and illustrates with an example, how a GODP can be instantiated and what the ontology ob-

tained by expanding the instantiation is. We introduce *foundational* GODPs by representing several ODPs from the literature, such as Role or Transition patterns in the style of DUL[2], the *DOLCE+DnS Ultralite* Upper Ontology [15], extending them by new specialisations of situations such as Scene, and Episode (Sect. 4). These are supplemented by GODPs for *data*, yielding quite an elaborate catalogue of interrelated, non-trivial GODPs (Sect. 5). In Sect. 6 the development process is adapted to the domain of service robotics, and in particular for modelling and storing episodes of activities. The end-user perspective is demonstrated in Sect. 7 with dedicated data patterns for Table Setting. Sect. 8 summarises coherence issues for GODPs: How can logically consistent episodes be achieved? What is a "good, well-performed" episode? How can episode records be easily and safely populated with data? Finally, the conclusion points out the lessons learned, and future extensions.

Thus the contribution of this paper is twofold:

- *consistent ontology engineering* with GODPs, demonstrated with an extensive case study: *episodes*, a significant extension to DUL patterns, and
- GODPs for *safe data population* by end-users revealing *substantial scaling*.

2. Motivation: Episodic Memories

GODPs and related techniques are very widely applicable in knowledge engineering; to illustrate their potential we will use a particular use case coming from the field of knowledge-driven service robotics. This use-case originates from the Collaborative Research Center EASE. In fact, service robotics provides opportunities for extensive use of heterogeneous combinations of reasoners [16]; in this paper we will focus on a particular topic: *narratively enabled episodic memories* (NEEMs). An episodic memory is a more or less comprehensive record of events that an agent observed; these may include raw sensor data and control values for actuators. A narratively-enabled record of events also contains interpretations of them: perception results, what tasks were being executed, what the results of the tasks were, possibly judgements of the outcomes according to some metric, etc.

The reason for having NEEMs is to provide semantically annotated data about real-world performance of routine household tasks: what was done, in what context, with what result. These data are subsequently used for teaching robots how to perform such tasks robustly and efficiently. A robot will accumulate and learn from NEEMs during its own active life, but also the life of other robots. However, NEEMs may also come from humans demonstrating a task—after all, humans are the best demonstrators of robust, efficient performance for household tasks.

We may already observe a complication: a NEEM cannot be a rigid list of contents. NEEMs may come from many kinds of agents, who will have different recording abilities; sometimes control values for actuators are available, sometimes not. Another complication is to formalize what is meant by an episode.

This results in a number of challenges for each of the stakeholders in the ontology engineering process:

- *ontology expert:* formalize basic concepts needed to represent NEEMs

[2] `http://www.ontologydesignpatterns.org/ont/dul/DUL.owl`

- *robotics expert:* define what kind of episodes may exist (e.g., what kinds of activities might be recorded in episodes), and what are acceptable structures for each kind of episode (e.g. tasks in the activity and constraints)
- *ontology user, NEEM creator:* populate a NEEM with data obeying structural and logical constraints on the kind of NEEM being generated
- *ontology user, NEEM consumer:* query a database of NEEMs by various criteria: kind of episode, agents/items involved, task success or failure, etc.

Throughout the rest of the paper, we will show how an ontology development process powered by GODPs will assist each stakeholder in meeting their challenges. We will present a motivating example for a *coherent* episode to give an overview of our approach, showing how to model a particular episode: setting a table for tea. The episode of setting a table for one person is abstracted to a pattern, re-used in an episode of setting the table for tea for two, sitting on opposite sides.

A distinction must be drawn between the modelling assumptions we have made in this paper and the GODP approach itself; GODPs can be applied to represent other modelling assumptions, e.g. situation modelling such as provided by Almeida [17]. In our case, we model episodes as sequences of transitions between scenes, both of which are DUL *Situations*. Scenes however correspond to situations that are considered unchanging in the absence of interventions from some agent; such interventions are the transitions. This implies that general knowledge about sequences must be formalized, as well as more domain-specific knowledge e.g. limits to what transitions are possible between which scenes.

We would like to set up dedicated development tasks for end-users such as fetchFrom[CrockeryCupboard][DessertPlate], transportTo[front][Table], place[on][Table] in a language they understand with a structured vocabulary to choose from. In the sequel we will describe the development stages to reach this goal. Due to lack of space, we will only present the most prevalent GODPs used; see [18] for more elaborate versions and a scaled-up example of more data input.

3. Generic Ontology Design Patterns in Generic DOL

The *Distributed Ontology, Model and Specification Language*, DOL, an OMG standard, allows the structured definition of ontologies with import, union, renaming, module extraction, etc. Thus, DOL is not "yet another ontology language", but a meta-language to define and manipulate ontologies; it may be used for a variety of ontology languages (e.g. OWL-DL) and logics in a heterogeneous manner.

Generic DOL [9,10,11] extends DOL by a parameterization mechanism for ontologies; this allows the expression of powerful semantic pre-conditions.

A brief description of Generic DOL can be given as follows. For the purposes of this paper, all ontologies will be OWL ontologies. A pattern has a name, a list of parameters, an optional imported ontology and a body. An instantiation of a pattern is made by giving a list of argument ontologies, one for each parameter. The parameters of a pattern are themselves ontologies or lists of ontologies. The axioms in a parameter are regarded as semantic pre-conditions that the argument of an instantiation of the pattern must fulfil. If the axioms are missing, no restrictions are imposed on the symbols of the parameter. The imported ontology provides the non-variable entities that are used for expressing these

```
pattern TASK_KINDS
[ Class: AncestorTask SubClassOf: Task;       %% ancestor in Task taxonomy
  Class: AncestorPre  SubClassOf: PreScene;   %% ancestor in Pre  taxonomy
  Class: AncestorPost SubClassOf: PostScene;  %% ancestor in Post taxonomy
  ObjectProperty: tAncestor;                  %% ancestor task relation
 {ObjectProperty: t} :: ts                  ] %% list of task relations
given Foundation =
 Class: Task[t] SubClassOf: AncestorTask     %% task with relation t
 Class: Pre[t]  SubClassOf: AncestorPre      %% pre-conditions for t
 Class: Post[t] SubClassOf: AncestorPost     %% goals of Task[t]
 ObjectProperty: t SubPropertyOf: tAncestor Domain: Pre[t] Range: Post[t]
then TASK_KINDS[AncestorTask; AncestorPre; AncestorPost; tAncestor; ts]
... Instantiation:
TASK_KINDS[Task[TableSetting]; Pre[TableSetting]; Post[TableSetting];
           transact[TableSetting]; [fetchFrom]                      ]
... DOL Expansion:
Class: Task[fetchFrom]       SubClassOf: Task[TableSetting]
Class: Pre[fetchFrom]        SubClassOf: Pre[TableSetting]
Class: Post[fetchFrom]       SubClassOf: Post[TableSetting]
ObjectProperty: fetchFrom    SubPropertyOf: transact[TableSetting]
   Domain: Pre[fetchFrom]    Range: Post[fetchFrom]
```

Figure 1. TASK_KINDS and expansion of instantiation

semantic pre-conditions. The body of the pattern, in the simplest unstructured variant, is an ontology using the symbols of the parameters. In general, any DOL structuring mechanism may be used in the body. If the pre-conditions given in the parameters hold for the arguments of an instantiation, a macro replacement substitutes the symbols of the arguments for the symbols of the parameters in the body. This gives rise to a structured DOL ontology that can be analyzed by Hets and flattened to an unstructured ontology, via an expansion procedure.

Fig. 1 contains a small example. The pattern TASK_KINDS allows the extension of an existing ontology with new tasks. A *task relation* t has pre- and post-conditions as domain Pre[t] and range Post[t], resp. In its body (after the "=" symbol), TASK_KINDS defines Task[t], Pre[t], and Post[t], and sets up their position in the taxonomy as subclasses of the corresponding ancestors, which are passed as parameters AncestorTask, AncestorPre, and AncestorPost; similarly, the object property t is defined as a subproperty of its ancestor tAncestor, with domain and range. The pattern defines both a property t and a concept Task[t], since it is convenient in some parts of the modelling to treat tasks as concepts to classify observed events, and in others as transformations between scenes.

A *parameterised name* such as Task[t] denotes a different name for each argument of t in an instantiation. At the end of the expansion process of instantiations in Generic DOL, parameterised names will be converted to proper OWL names by *stratification*[3]: if t is given the argument fetchFrom, then Task[t] expands to Task[fetchFrom], and after stratification to Task_fetchFrom.

The definitions for t are iterated over a list of such task relations. The list is given as a parameter in a constructive fashion t :: ts stating the head of the list, t, the concatenation

[3] "[" and "," become underscores "_", "]" are deleted.

```
pattern FUNCTION_INVERSE
 [ObjectProperty: f; Class: D; Class: R; ObjectProperty: finv] =
ObjectProperty: f Domain: D Range: R %% Characteristics Functional
ObjectProperty: finv Domain: R Range: D InverseOf: f
```

Figure 2. FUNCTION_INVERSE (see text for **Characteristics Functional**)

```
pattern SEQUENCE [Class: E] %% kind of sequence elements
=     FUNCTION_INVERSE[hasLast[E];      Seq[E]; E; isLastOf[E]  ]
 and FUNCTION_INVERSE[hasFirst[E];     Seq[E]; E; isFirstOf[E] ]
 and FUNCTION_INVERSE[isSeqElemOf[E]; E; Seq[E]; hasSeqElem[E]]
 and FUNCTION_INVERSE[succ[E];         E;       E; prec[E]       ]

pattern ROLE [ Class: Rle; Class: Ancestor;
 Class: Performer; ObjectProperty: performedBy; ObjectProperty: performs;
?Class: Provider; ?ObjectProperty: providedBy; ?ObjectProperty: provides]
=     FUNCTION_INVERSE[performedBy; Rle; Performer; performs]
 and FUNCTION_INVERSE[providedBy;  Rle; Provider;  provides]
then Class: Rle SubClassOf: Ancestor,
                performedBy some Performer, providedBy some Provider
```

Figure 3. SEQUENCE and ROLE

symbol, "::", and the tail of the list, ts. The instantiation of TASK_KINDS at the end of the body effects a recursion, with the same arguments as the parameters, except for the tail of the list, ts, as the last argument.

Fig. 1 shows the expansion of an instantiation of TASK_KINDS, taken from Fig. 9 below. The arguments are again parameterised names; the last argument denotes a singleton list [fetchFrom].

The expressions in the body of the ontology TASK_KINDS are in OWL Manchester Syntax. The phrase A **then** B in DOL indicates that all definitions in A are visible in B, where A and B are flat (OWL) ontologies or instantiations of GODPs. Similarly, A **and** B is the union of the two ontologies.

The parameter AncestorTask is constrained by an axiom: it must be a subclass of Task. Since the subclass property is interpreted in OWL to be subset, this means in fact that AncestorTask must be "somewhere in the subclass chain" ending in Task in the taxonomy. Task in turn is defined in the imported (**given**) ontology Foundation[4]. This demonstrates additional *structural consistency* for ontology extension; it may be compared with *strong typing* in programming languages, and we will refer to it as *"typing"* at the model level. Such semantic conditions are not restricted to subclass axioms: arbitrarily complex OWL assertions may be used.

The pattern FUNCTION_INVERSE in Fig. 2 is intended for the extension of an existing ontology by declaring a new object property, a function f, whose name is provided as the first parameter, and its domain and range classes as second and third parameter,

[4] Foundation includes (a selective view to) DUL, e.g. Task, and extensions, e.g. PreScene.

resp. Moreover, the name of its inverse function finv is given as the fourth parameter. The body defines f and finv with domains and ranges.

Confinement of Design Choices. The choice of the way in which to state that f should be a function is strictly confined to the body of FUNCTION_INVERSE. The **Functional** characteristics for f is only provided as a comment; this becomes necessary since OWL-DL explicitly forbids this characteristics for a function that is a superproperty of a property chain. As soon as we needed such axioms, we adjusted the pattern once, for all instantiations, to accommodate the restriction; only this one pattern needs readjustment for a more powerful reasoner.[5]

The pattern SEQUENCE in Fig. 3 defines (linear) sequences Seq[E], a collection with functions succ[E] and its inverse prec[E] between elements. The parameter E denotes the kind of sequence elements; sequences are distinguished using [E] in parameterised names: Seq[E] (succ[E] etc.) are different in each instantiation. The pattern FUNCTION_INVERSE is instantiated several times in the body of SEQUENCE; this provides a clear definition with good structuring.

Roles. The foundational ODP for *role* in DUL has received considerable attention in the literature, see [12,11] for a comparison of different approaches. With ROLE in Fig. 3, we use a GODP in a version that allows different choices of names for the object properties performs and provides, and their inverses. We believe that this is more suggestive in most applications than using inheritance of these standard names, and alleviates the need for a careful scoping approach used in conventional ODP approaches, overloading such names for many applications.[6]

ROLE allows the parameters Provider, as well as providedBy and provides, to be optional, indicated by the question mark. Thus the pattern may also be used just for creating different "manifestations" of a Performer.[7] However, we realised how important the notion of context using a Provider is (cf. transitions in Sect. 4).

4. Ontology Expert Perspective: Foundational Patterns

An *episode*[8] is a sequence of task executions; it corresponds to linear "unrolling" of a plan (that has branches and iterations).[9] In the pattern EPISODE (Fig. 4) we model an Episode[E] as a Sequence[E] of Transition[E]s between Scene[E]s using instantiations of the patterns SEQUENCE, TRANSITION and SCENE.[10]

[5] See discussion in [11]. One could add a functionality axiom in a more powerful logic in a heterogeneous context, at the price of losing decidability of OWL-DL reasoners enforcing the restriction (`https://www.w3.org/TR/owl2-syntax/#Global_Restrictions_on_Axioms_in_OWL_2_D`).

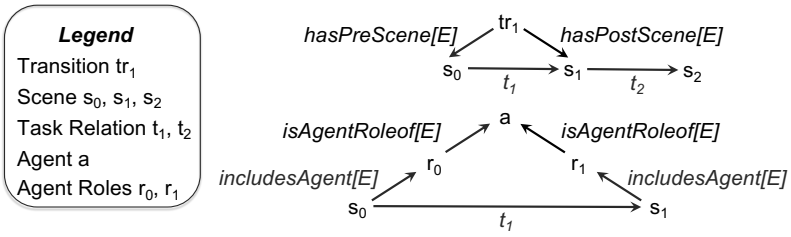
[6] We are using a differentiated approach to inheritance (cf. also [12]): R is tied to its AncestorRole in the taxonomic hierarchy of Role in DUL, while object properties in ROLE or other patterns are intentionally kept separate, i.e. not related to ancestor relations or overloaded.

[7] The notion of time is irrelevant here (TEMPORAL_Extent[Rle] is omitted); cf. a discussion in [19,20,11]. DUL uses the notions isClassifiedBy for performs, and defines for provides.

[8] "episode" and "scene" are new terms, specialisations of DUL situations.

[9] As in ROLE, we have no need for the notion of time; sequence elements (or manifestations in roles, cf. [19,20]) may however be decorated with time intervals in the context of actions.

[10] We use roles explicitly: e.g. includesAgent[E] has AgentRole[E] as domain and not XAgent




```
pattern EPISODE [ Class: E;                    %% kind of episode
  Class: XAgent SubClassOf: PhysicalAgent;     %% kind of agents
  Class: XItem  SubClassOf: PhysicalObject;    %% kind of items
  Class: XEnv   SubClassOf: PhysicalObject]    %% kind of environment
given Foundation =
 SCENE[E;XAgent;XItem;XEnv] and TRANSITION[E] and SEQUENCE[Transition[E]]
then Class: Episode[E] SubClassOf: Seq[Transition[E]], Episode

pattern SCENE [ Class: E;                      %% kind of episode
  Class: XAgent SubClassOf: PhysicalAgent;     %% kind of agents
  Class: XItem  SubClassOf: PhysicalObject;    %% kind of items
  Class: XEnv   SubClassOf: PhysicalObject]    %% kind of environment
given Foundation %% PhysicalObject, PhysicalAgent, Scene, Transition
=     Class: Scene[E] SubClassOf: Scene
then ROLE[AgentRole[E];    AgentRoleScene;
      XAgent;    isAgentRoleOf[E];    hasAgentRole[E];
      Scene[E]; isAgentIn[E];        includesAgent[E]]
 and %% ... analogously for EnvRole[E], ItemRole[E]
```

Figure 4. EPISODE and SCENE

```
pattern TRANSITION [ Class: E ] given Foundation =
ROLE[Transition[E]; Transition; Task[E]; executesTask[E]; executedIn[E];
                                Pre[E];  hasPreScene[E]; isPreSceneOf[E]]
and FUNCTION_INVERSE[hasPostScene[E]; Transition[E];
                     Post[E]; isPostSceneOf[E]      ] then
 Class: Scene[E]       SubClassOf: Scene
 Class: Pre[E]         SubClassOf: Scene[E], PreScene
 Class: Post[E]        SubClassOf: Scene[E], PostScene
 Class: Transition[E]  SubClassOf: Transition, hasPostScene[E] some Post[E]
 Class: Task[E]        SubClassOf: Task
 ObjectProperty: transact[E] Domain: Pre[E] Range: Post[E]
      SubPropertyOf: transact
```

Figure 5. TRANSITION

The parameter E denotes the kind of episode in distinctive parameterised names; it will be specialised to a particular domain scenario below (Sect. 6). E is passed as argument to instantiations in the body, as are the other parameters.

A *scene* is a snapshot of a part of the state of the world under consideration in an episode of kind E. Consider SCENE in Fig. 4, and note the applications of the ROLE pattern: a Scene[E] includesAgent[E] an agent of kind XAgent in its role Agent-

```
pattern DATA_Role [ Class: R SubClassOf: Role;
  ObjectProperty: performs; ObjectProperty: providedBy;
  Individual: perf; Individual: prov; Individual: rle ]
given Foundation =
 Individual: rle  Types: R Facts: providedBy prov
 Individual: perf          Facts: performs rle
```

Figure 6. DATA_Role

```
pattern DATA_Scene [ Class: E;                    %% kind of episode
  Class: XAgent    SubClassOf: PhysicalAgent;     %% kind of agent
  Class: XEnv      SubClassOf: PhysicalObject;    %% kind of environment
  Individual: s;                                  %% scene id
  Individual: a    Types: XAgent;                 %% agent of episode
  Individual: env  Types: XEnv                  ] %% environment of episode
given Foundation %% PhysicalObject, PhysicalAgent
=   DATA_Role[AgentRole[E]; hasAgentRole[E]; isAgentIn[E]; a; s; a[s]  ]
and DATA_Role[EnvRole[E];   hasEnvRole[E];   isEnvIn[E]; env; s; env[s]]
```

Figure 7. DATA_Scene

Role[E]; analogously an environment of kind XEnv in its EnvRole[E], and a special object of kind XItem in its ItemRole[E] as its major focus (see [18]).

A *transition* Transition[E] (Fig. 5) corresponds to a mapping from a Scene[E], its pre-scene Pre[E], to another, its post-scene Post[E].

A task assumes a role in a transition: a task Task[E] is executedIn[E] a Transition[E], which hasPreScene[E] Pre[E] (acting as perfomer and provider, resp.); thus a transition has its pre-scene as context (also its post-scene by hasPostScene[E]). A *task relation* t maps from a pre-scene Pre[t] to a post-scene Post[t]; in fact, this relation governs the transition the task is executed in and thus becomes the *transition relation*.[11] Pre[t], Post[t] express pre- and post-conditions, cf. Fig. 1.[12]

5. Ontology Expert Perspective: Data Patterns

As we have seen so far, GODPs may be used to extend an existing ontology in a controlled, structured way with additional safeguards such as "typing". We will now demonstrate how such safer change management may also be applied to data, to guard their typing (with **Types**, e.g. on input), and to generate their intricate interrelations (e.g. with **Facts**).

Assuming that roles have been set up by instantiating the ROLE pattern (Fig. 3), then instantiations may be obtained using DATA_Role, see Fig. 6; cf. the instantiations of DATA_Role for the agent and the environment in DATA_Scene (Fig. 7). We rely heav-

[11] Pre[E] and Post[E] are the domain and range of a relation transact[E] (cf. [18]), the transition super-relation for all individual relations t for episodes of kind E, cf. Fig. 9.

[12] Task[t] is a kind of reification of t. This correspondence cannot be expressed in OWL, but since t, Task[t] etc. are synchronously generated by one GODP, they exist coherently.

```
ontology EASE_TableSetting_log = Foundation then
  Class: Agent[TableSetting] SubClassOf: NaturalPerson or AutonomousRobot
then EPISODE[TableSetting;Agent[TableSetting];Tableware;DesignedContainer]

pattern DATA_Episode_TableSetting[Individual: e; Individual: a;
   Individual: env; Individual: s0; {ObjectProperty: t0} :: ts]
 given EASE_TableSetting_log =
DATA_Episode[TableSetting; Agent[TableSetting]; DesignedContainer;
             e; a; env; s0; t0 :: ts]

pattern DATA_Initial_TableSetting
  [ Individual: s0; Individual: a0; Individual: env0 ]
 given EASE_TableSetting_log =
DATA_Scene[TableSetting;Agent[TableSetting];DesignedContainer;s0;a0;env0]
```

Figure 8. EASE_TableSetting_log, DATA_Episode_TableSetting, DATA_Initial_TableSetting

ily on parameterised names (cf. Sect. 3): the individuals a[s] or env[s] in the instantiations are parameterised by the (pre)scene s, i.e. the provider is included as a context in the name; this distinguishes the role instance from others (for the same performer) in different situations.

Consider the analogous application to the role Transition[E]: the "performer" should be the task relation t (executedIn[E] Task[t]); t, however, is not an individual but an object property. Therefore a pattern DATA_RoleTransition has been devised in analogy to DATA_Role (see [18]). The role instance is tied to its PreScene sPre; thus each transition instance is different due to a different sPre.

The intricate patterns DATA_Episode, DATA_Transition in [18] demonstrate the inherent complexity of the data interrelations, localised and thus manageable.

6. Domain Expert Perspective: Configuration Patterns

In Fig. 8 the pattern EPISODE is instantiated to the Table Setting scenario. We specify, what kind of agent, items, environment, and tasks are admissible in this scenario: e.g. for Agent[TableSetting], only items of class NaturalPerson or AutonomousRobot are allowed; this could be made more specific, if desired. These constraints are passed along in the parameters and give rise to corresponding checks for "typing". Consider e.g. the specialisation of DATA_Episode to DATA_Episode_TableSetting: DATA_Episode is partially instantiated by these constraints as arguments (while others such as a or env are still left as parameters), such that future instantiations of DATA_Episode via DATA_Episode_TableSetting will e.g. require agents a to be of **Types** XAgent, i.e. Agent[TableSetting] (cf. Fig. 8, and SCENE in Fig. 4 above).

Other patterns are analogously specialised to this scenario, such as the set-up of the initial scene in DATA_Initial_TableSetting (Fig. 8). Similarly, we may configure other scenarios, e.g. for cooking in a Kitchen with Cookware etc.[13]

[13] For other application scenarios, more objects in focus like Item will have to be introduced, e.g. the cooking container or preparation device, or the cooking utensil as a tool.

```
pattern TASK_KIND_ItemS
[ Class: XItem SubClassOf: PhysicalObject; %% kind of item
 {ObjectProperty: t} :: tS;                 %% ancestor task relations
 {Class: A  SubClassOf: XItem} :: As      ] %% list of items
given Foundation =
let pattern TASK_SubKindS [ Class: B :: Bs ] =
      TASK_KINDS[Task[t]; Pre[t]; Post[t]; t; [t[B]]] and TASK_SubKindS[Bs]
in  TASK_SubKindS[A :: As] and TASK_KIND_ItemS[XItem; tS; A :: As]

ontology EASE_Data_Task_TeaForTwo_log = EASE_TableSetting_log
and TASK_KINDS[Task[TableSetting]; Pre[TableSetting]; Post[TableSetting];
      transact[TableSetting];  [fetchFrom, transportTo, place]          ]
and TASK_KINDS_Spatial[SpatialRelation3D; [transportTo]; [front, back] ]
and TASK_KINDS_Spatial[SpatialRelation3D; [place];       [on, topright]]
and TASK_KIND_ItemS[DesignedContainer; [fetchFrom]; [CrockeryCupboard]]
and TASK_KIND_ItemS[DesignedContainer;
      [transportTo[front], transportTo[back]]; [Table]]
and TASK_KIND_ItemS[DesignedContainer; [place[on]]; [Table] ]
and TASK_KIND_ItemS[Crockery;          [place[on]]; [Saucer]]
```

Figure 9. TASK_KIND_ItemS and Log of Tasks for TableSetting

7. End-User Perspective: Dedicated Data Patterns

In the same way, we are now able to set up dedicated development tasks for the end-user. In EASE_Data_Task_TeaForTwo_log a scenario for Table Setting is initialised providing the necessary tasks (cf. Sect. 6); in Fig. 9, a hierarchy of vocabulary is configured for increasingly specialised operations, such as

transportTo
transportTo[front] transportTo[back]
transportTo[front][Table] transportTo[back][Table]

from which the user may choose appropriate ones when annotating an episode or when defining a coherent episode template. Fig. 9 not only defines the task vocabulary, but also configures and constrains e.g. the allowed spatial relations. The larger example at [18] with Cutlery, a CutleryDrawer, etc., includes also the requisite PastryFork and TeaSpoon; and tasks such as place[top][DessertPlate][PastryFork].

The pattern TASK_Kinds (see Fig. 1) declares corresponding information for a given list of task relations, and links it to ancestors in the resp. hierarchies. Similarly, TASK_KIND_ItemS (Fig. 9) creates sub-relations for a list of items, e.g., fetchFrom[CrockeryCupboard]. TASK_KIND_SpatialRelationS analogously creates similar sub-relations for spatial relations, e.g. transportTo[front], place[topright].

Parameterised Episode Tea For One. The vocabulary of tasks is utilised in the pattern DATA_Episode_TeaForOne (Fig. 10); e.g. transportTo[spr][Table] denotes a task for transporting an item to the table in position spr; spr is the last parameter of DATA_Episode_TeaForOne, denoting a spatial position.

Episode Tea For Two. DATA_Initial_TableSetting in Episode_TeaForTwo_log (Fig. 10), sets up BKB as agent, BKBsTea_s0 as initial scene, and EASE_Lab as environment. Two instantiations of DATA_Episode_TeaForOne result in episodes BKBsTeaForOne_front and BKBsTeaForOne_back, as indicated by their first arguments;

```
pattern DATA_Episode_TeaForOne
 [ Individual: e; Individual: a; Individual: env; Individual: s0;
   Individual: spr Types: SpatialRelation3D ] %% spatial position
 given EASE_Data_Task_TeaForTwo_log =
DATA_Episode_TableSetting [ e; a; env; s0;
 [fetchFrom[CrockeryCupboard][DessertPlate], transportTo[spr][Table],
                                             place[on][Table],
  fetchFrom[CrockeryCupboard][Saucer],       transportTo[spr][Table],
                                             place[topright][DessertPlate],
  fetchFrom[CrockeryCupboard][TeaCup],       transportTo[spr][Table],
                                             place[on][Saucer]          ] ]

ontology Episode_TeaForTwo_log = EASE_Data_Task_TeaForTwo_log
and DATA_Initial_TableSetting[BKBsTea_s0; BKB; EASE_Lab ]
and DATA_Episode_TeaForOne[BKBsTeaForOne_front;
     BKB; EASE_Lab; BKBsTea_s0;                  front ]
and DATA_Episode_TeaForOne[BKBsTeaForOne_back;
     BKB; EASE_Lab; post[BKBsTeaForOne_front]; back  ]
and CONC_Episodes[TableSetting; Episode[TableSetting];
     BKBsTeaForOne_front; BKBsTeaForOne_back; BKBsTeaForTwo ]
```

Figure 10. DATA_Episode_TeaForOne and Instantiations in Episode_TeaForTwo_log

their last arguments, front and back, denote the corresponding position on the table; the initial scene BKBsTea_s0 and the post-scene of the first episode are given as further arguments. Finally, the two episodes are concatenated with the pattern CONC_Episodes to the complete episode BKBsTeaForTwo (see [18]).

The expanded instantiations of the patterns in the examples have been successfully checked for consistency with standard OWL reasoners.

8. Episode Data Consistency

We will now look at how the proposed GODP based approach assists the various stakeholders in an ontology engineering process to create and maintain a consistent, well-structured database. For our use case, this is a database of episodes recording agents' activities in the household domain. We use "consistency" in the "semantic integrity" sense from database research [21]: the data we have about an episode describe an episode that could have happened. There are constraints on what is a *logically coherent* episode that must be enforced upon the data, either via structuring how episodes can be added to the database, or by checking conformance of new episodes to formalized specifications.

Logically Consistent Episodes from an Ontology and Domain Expert Perspective. Logical constraints on the structure of an episode come from the definition of an Episode; records not obeying such constraints cannot correspond to any possible episode and thus must contain some error. We defined episodes as linear sequences of transitions between scenes: thus a transition cannot be its own successor, even indirectly. We show how to enforce this in Sect. 4, and Sect. 5 for data. Episodes may also relate to each other (e.g. by specialisation when instantiating a pattern, by concatenation, refinement, projection, etc.), imposing further constraints. Constraints may also be more

domain specific: e.g. for a robot doing manipulation, certain actions are only available for objects it has in hand.

The more extensive example [18] includes an additional role for items in the environment; it illustrates how the roles an item plays change depending on whether it is manipulated by the robot during a task, or placed again into the background. Constraints preserve objects between pre- and post-scenes, e.g. a fetch task can only affect an object already present in the environment in its pre-scene, and can only place that same object in its post-scene at some location. The fact that the individual item participating in the pre-scene is identical to one in the post-scene cannot be expressed in OWL, requiring a heterogeneous solution.

All episodes in a database must obey logical constraints on episode structure, achieved by restricting how data are entered, e.g. entry forms that impose constraints of identity, and/or checking data before adding it to the database.

Domain Expert Perspective: "Well-Performed" Episodes. Closely related to consistency is the question whether an episode describing activity performance is completed successfully. Criteria for this are domain and task specific, but can often be logically described. A transport task is successful, if it ends with the transported item at the target location. We can also express the constraint that a transportTo task should maintain hasVerticalOrientation for the handled item, important when transporting a drink in a cup!

Episodes of activities that do not follow "good performance" constraints are not necessarily invalid; in fact it may be very interesting to keep episodes of activities performed poorly in a robot's database as well, if only to learn from mistakes. However, it becomes important in such cases to distinguish what is a mistake or not, hence it must be possible to query whether stored episodes obey some constraints on what a well-performed activity of a particular kind looks like.

We have shown how such constraints can be formalized and used via GODPs. It will be a subject for future work how to treat episodes of failure precisely.

End-User Perspective: Easy and Safe Population of Episode Records. We ensure the *coherence and consistency of data* through logical modelling of concepts by setting up the interrelation of data and checking their *consistency upon data entry*. The end-user should be relieved from such details as much as possible, to focus on the task at hand, while safe population with data is assured.

9. Conclusion

The robotic episodic memory provides a very interesting testbed for ontology and database solutions. On the one hand, the task itself of implementing such an episodic memory is complex, for reasons pertaining to the heterogeneity of contents that is caused by the variety of sensor and actuator combinations available on robots, to the logical characterization of what is an episode, and to the formal description of what is a logically sane, or qualitatively good, episode of a particular kind. On the other hand, the people who work with such robotic episodic memories have different goals and different levels of experience with ontological modelling, and in particular the users of such a knowledge-driven system, robotics engineers, will need as much as possible of the inferential and consistency checking work to be encoded in the ontology itself.

To this end, we have implemented GODPs to model episodes, scenes, and transitions, in such a way that constraints on logical consistency or task conformity are encoded in the patterns, and constrain what can be entered into a database of episodes to avoid error. These patterns also allow definitions of sanity checks on already existing databases. Our discussion has been organized by looking at what the different stakeholders in an ontology development process—ontology experts, domain experts, end users—are expected to contribute to the process and how the GODPs benefit them. Foundational GODPs created by ontology experts constrain domain-specific ontological extensions, which in turn constrain the data that end users manipulate, allowing each group of stakeholders to focus on their own expertise and aims. These foundational GODPs generalise from episodes of a particular kind, and allow safe specialisation (cf. Sect. 6).

We would like to emphasise that we gained a lot of insight into the proper modelling of foundational concepts from DUL and its associated ODPs. We believe that GODPs make such ODPs practical; thus GODPs provide means for specifying patterns for (large amounts of) *data* and manifold, extensive applications.

GODPs share many objectives with *Parameterised OTTR Templates* with macro expansion [22,23,24]. Generic DOL provides more comprehensive list parameters with recursion, parameterised names, and ontology parameter constraints.

The pattern ROLE is ubiquitously used in this paper. It allows for good structuring; its use to tie in tasks with transitions (cf. Sect, 4 and 5) was a revelation even for us: the pre-scene provides the necessary distinctive context to define a specialised (data) manifestation of a task in its role when executed in a transition.

The patterns EPISODE, TRANSITION, SCENE, and in particular their data counterparts, show the diligence and inherent complication when taking *data* seriously in modelling. One cannot avoid the distinction between objects *and* their diverse roles.[14] On the other hand it is no wonder that this is often glossed over, since the modelling becomes rather intricate and large—it seems quite impossible to manage without the generative approach of GODPs.

It was very instructive to see during our own development, how the different levels of checking assist the debugging (which was quite extensive for this rather complicated example): at the first level, the structural check for mismatch of parameter versus argument kinds (e.g. Individual instead of Class, or vice versa) occurred on the order of 20 times. Most notably, an interesting and quite fundamental oversight was uncovered this way: originally, DATA_Role was instantiated in DATA_Transition, where the object property t was mistakenly used as an Individual parameter; this led to the introduction of the analogous pattern DATA_RoleTransition, cf. Sect. 5.

It will be an interesting future research issue to investigate, to what extent consistency checks, expressed as pre-conditions on parameters generating proof obligations on arguments, may be delegated to the development phase and become obsolete once a coherent modelling has been obtained (including data patterns).[15] Then one might want to move axioms, which "only" relate to consistency, from the body to parameters in Generic DOL as much as possible (cf. [11]); this view offers interesting perspectives for

[14] We took some inspiration from [19,20], realising eventually that their notion of "manifestations" is in effect an application of the ROLE pattern.

[15] Proof obligations generated by instantiations are checked with an automated OWL reasoner. The actual proof may be deferred; then the expansion is generated but may be ill-formed because semantic pre-conditions do not hold; a change requires to redo the proofs.

efficiency of deduction in "production runs", say in robotics applications, helped by the use of GODPs.

More applications, e.g. for configuration (cf. [10]) or the cooking domain (cf. [8]), will follow. A recipe is a workflow, essentially a prescription for an episode in which it is executed. We expect a workflow to be a rather straightforward generalisation of an episode to a directed graph of transitions, where the function succ is generalised to a relation next.

Safe Scaling of Data. The modelling of the example has been a non-trivial and sizable effort. The extended example (see [18]) generates ca. 800 axioms in OWL, ca. 1 axiom per dense line of original Generic DOL (not counting ca. 450 lines of Foundation). On this basis, 15 tasks (i.e. 6 more than the 3*3 in DATA_Episode_TeaForOne and Episode_TeaForTwo_log (Fig. 10)) generate ca. 750 axioms and 165 individuals, thus about 50 axioms each; we can expect a similar scaling factor for each task. It seems quite impossible to effect an error-free construction by hand, whereas the input of such tasks can be entrusted to an end-user with a suitable interaction interface (e.g. for DATA_Episode_TeaForOne).

It will be interesting to see in a (planned) user study, how domain experts cope with ready-made GODPs, and end-users with GODPs supporting data input.

Acknowledgments. We are grateful to Till Mossakowski and Fabian Neuhaus for their cooperation in the development of Generic DOL, and Daniel Beßler and Robert Porzel for their suggestions regarding ontologies in the robotics domain.

References

[1] Blomqvist E, Sandkuhl K. Patterns in Ontology Engineering: Classification of Ontology Patterns. In: Chen C, Filipe J, Seruca I, et al., editors. ICEIS 2005, Proceedings of the Seventh International Conference on Enterprise Information Systems, Miami, USA, May 25-28, 2005; 2005. p. 413–416.

[2] Gangemi A. Ontology Design Patterns for Semantic Web Content. In: Gil Y, Motta E, Benjamins VR, et al., editors. ISWC 2005; (LNCS; Vol. 3729). Springer; 2005. p. 262–276.

[3] Presutti V, Gangemi A. Content Ontology Design Patterns as Practical Building Blocks for Web Ontologies. In: Li Q, Spaccapietra S, Yu ESK, et al., editors. Conceptual Modeling - ER 2008, 27th International Conference on Conceptual Modeling, Barcelona, Spain, October 20-24, 2008. Proceedings; (Lecture Notes in Computer Science; Vol. 5231). Springer; 2008. p. 128–141. Available from: https://doi.org/10.1007/978-3-540-87877-3_11.

[4] Hitzler P, Gangemi A, Janowicz K, et al., editors. Ontology Engineering with Ontology Design Patterns - Foundations and Applications. (Studies on the Semantic Web; Vol. 25). IOS Press; 2016.

[5] Hammar K, Hitzler P, Krisnadhi A, et al., editors. Advances in Ontology Design and Patterns [revised and extended versions of the papers presented at the 7th edition of the Workshop on Ontology and Semantic Web Patterns, WOP@ISWC 2016, Kobe, Japan, 18th October 2016]; (Studies on the Semantic Web; Vol. 32). IOS Press; 2017. Available from: http://ebooks.iospress.nl/volume/advances-in-ontology-design-and-patterns.

[6] Shimizu C, Hirt Q, Hitzler P. MODL: A Modular Ontology Design Library. In: Janowicz K, Krisnadhi AA, Poveda-Villalón M, et al., editors. Proceedings of the 10th Workshop on Ontology Design and Patterns (WOP 2019) co-located with 18th International Semantic Web Conference (ISWC 2019), Auckland, New Zealand, October 27, 2019; (CEUR Workshop Proceedings; Vol. 2459). CEUR-WS.org; 2019. p. 47–58. Available from: http://ceur-ws.org/Vol-2459/paper4.pdf.

[7] Shimizu C, Krisnadhi A, Hitzler P. Modular Ontology Modeling: a Tutorial. In: Cota G, Daquino M, Pozzato GL, editors. Applications and Practices in Ontology Design, Extraction, and Reasoning. IOS Press; 2020. Studies on the Semantic Web. In press.

[8] Krieg-Brückner B. Generic Ontology Design Patterns: Qualitatively Graded Configuration. In: Lehner F, Fteimi N, editors. KSEM 2016, The 9th International Conference on Knowledge Science, Engineering and Management; (Lecture Notes in Artificial Intelligence; Vol. 9983). Springer International Publishing; 2016. p. 580–595.

[9] Codescu M, Krieg-Brückner B, Mossakowski T. Extensions of Generic DOL for Generic Ontology Design Patterns. In: Barton A, Seppälä S, Porello D, editors. Proceedings of the Joint Ontology Workshops 2017 Episode V: The Styrian Autumn of Ontology, September 23-25, Graz, Austria. CEUR-WS.org; 2019. CEUR Workshop Proceedings. Available from: http://ceur-ws.org/.

[10] Krieg-Brückner B, Codescu M. Deducing Qualitative Capabilities with Generic Ontology Design Patterns. In: Silva MF, Lima JL, Reis LP, et al., editors. Robot 2019: Fourth Iberian Robotics Conference. Advances in Robotics, Volume 1.; Faculty of Engineering, University of Porto. Springer Nature Switzerland AG, Cham; 2020. p. 391–403; (Advances in Intelligent Systems and Computing, AISC; 1092). Available from: https://doi.org/10.1007/978-3-030-35990-4_32.

[11] Krieg-Brückner B, Mossakowski T, Codescu M. Generic Ontology Design Patterns: Roles and Change Over Time. In: Advances in Pattern-based Ontology Engineering. Amsterdam: IOS Press; to appear.

[12] Krieg-Brückner B, Mossakowski T, Neuhaus F. Generic Ontology Design Patterns at Work. In: Barton A, Seppälä S, Porello D, editors. Proceedings of the Joint Ontology Workshops 2019 Episode V: The Styrian Autumn of Ontology, Graz, Austria, September 23-25, 2019; (CEUR Workshop Proceedings; Vol. 2518). CEUR-WS.org; 2019. Available from: http://ceur-ws.org/Vol-2518/paper-BOG2.pdf.

[13] Object Management Group. The Distributed Ontology, Modeling, and Specification Language (DOL) ; 2016. OMG standard at omg.org/spec/DOL. See also dol-omg.org.

[14] Mossakowski T, Maeder C, Lüttich K. The Heterogeneous Tool Set, Hets. In: Grumberg O, Huth M, editors. TACAS 2007; (LNCS; Vol. 4424). Springer; 2007. p. 519–522.

[15] Presutti V, Gangemi A. Dolce+D&S Ultralite and its Main Ontology Design Patterns. In: Hitzler P, Gangemi A, Janowicz K, et al., editors. Ontology Engineering with Ontology Design Patterns - Foundations and Applications. (Studies on the Semantic Web; Vol. 25). IOS Press; 2016. p. 81–103. Available from: https://doi.org/10.3233/978-1-61499-676-7-81.

[16] Bateman J, Beetz M, Beßler D, et al. Heterogeneous Ontologies and Hybrid Reasoning for Service Robotics: The EASE Framework. In: Ollero A, Sanfeliu A, Montano L, et al., editors. ROBOT 2017:

Third Iberian Robotics Conference - Volume 1, Seville, Spain, November 22-24, 2017; (Advances in Intelligent Systems and Computing; Vol. 693). Springer; 2017. p. 417–428. Available from: https://doi.org/10.1007/978-3-319-70833-1_34.

[17] Almeida J, Costa P, Guizzardi G. Towards an Ontology of Scenes and Situations. In: 2018 IEEE Conference on Cognitive and Computational Aspects of Situation Management (CogSIMA). IEEE; 2018. p. 29–10.

[18] Krieg-Brückner B, Codescu M, Pomarlan M. Complete Example Repository ; 2020. Available from: https://ontohub.org/repositories/neem-godps.

[19] Katsumi M, Fox M. A Logical Design Pattern for Representing Change Over Time in OWL. In: Blomqvist E, Corcho O, Horridge M, et al., editors. 8th workshop on ontology design patterns - wop 2017; 2017. Available from: http://ontologydesignpatterns.org/wiki/images/4/42/Paper-05.pdf.

[20] Katsumi M, Fox M. A Logical Design Pattern for Representing Change Over Time: Applications in Transportation Planning. In: Advances in Pattern-based Ontology Engineering. Amsterdam: IOS Press; to appear.

[21] Bhargava B, Lilien L. Enforcement of data consistency in database systems. Sadhana. 1987 10;11:49–80.

[22] Skjæveland MG, Forssell H, Klüwer JW, et al. Pattern-Based Ontology Design and Instantiation with Reasonable Ontology Templates. In: Blomqvist E, Corcho O, Horridge M, et al., editors. 8th workshop on ontology design patterns - wop 2017; 2017. Available from: http://ontologydesignpatterns.org/wiki/images/6/66/Paper-04.pdf.

[23] Skjæveland MG, Lupp DP, Karlsen LH, et al. Practical ontology pattern instantiation, discovery, and maintenance with reasonable ontology templates. In: Vrandecic D, Bontcheva K, Suárez-Figueroa MC, et al., editors. The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, Monterey, CA, USA, October 8-12, 2018, Proceedings, Part I; (Lecture Notes in Computer Science; Vol. 11136). Springer; 2018. p. 477–494. Available from: https://doi.org/10.1007/978-3-030-00671-6_28.

[24] Skjæveland MG, Forssell H, Klüwer JW, et al. Pattern-Based Ontology Design and Instantiation with Reasonable Ontology Templates. In: Advances in Pattern-based Ontology Engineering. Amsterdam: IOS Press; to appear.