From Static Description of States to Dynamic Processes: Complex Processes Modeled as a Combination of Simpler Mechanisms

Andrew U. Frank & Amin Abdalla

June 29, 2012

Institute for Geoinformation
Vienna University of Technology

Abstract

Modern software engineering, especially for business applications, is process oriented. A focus on process in ontology is therefore warranted. The specification of basic processes with algebraic specifications in the ADT style are straight forward but the approach fails practically for realistically complex processes. The novel approach shown here is the assumption that *complex processes are the result of combination of basic processes;* the static combintaiotn of states in an ontology translates then directly to the combination of the mechanism which apply changes to the state. Two combination methods are identified: products of states which lead to process descriptions as sum of processes, and indexed set of states, which lead to process descriptions as products of index and processes. The method is shown here with an example of a multi-step process to prepare and get a passport from an state office. The ontology of the process can be extended in a principled way to include the information necessary for the automatic derivation of an optimal plan, as shown in the last part of the paper.

Keywords: process-ontology, mechanisms, image-schemata, haskell

1 Introduction

Modern software engineering, especially for business applications, is process oriented; a focus on process in ontology is therefore warranted. The description of basic processes with algebraic specifications as abstract data types is well researched and documented [8][7]; it is based on the representation of the effects of one operation applied to the result of another operation and formally specifies

an (universal) algebra up to isomorphism. We use the word *mechanism* for a set of operations together with an algebraic axioms which affect the same state and are defined in one abstract data type (ADT) specification. For example, a *container* with operations *putIn* and *takeOut* to change the state of the container and observers like *isEmpty* and *contains*; the effect of the operations are defined by axioms like $contains(a, putIn(a, c)) = true$.

The ADT approach does not scale and fails practically for realistically complex processes. We posit that complex processes result from the combination of basic processes and that only few principled ways of combination are necessary [4]. The example used here is an agent in a world consisting of his home, stores, offices and banks . The agent considers the multi-step process to prepare and get a passport from a state office. The static ontology of the world is a collection of things which have a state. This paper shows how this static description of an agent and a world as a collection of locations with shops etc. and paths for movement between the location gives a blueprint for the dynamic model of processes. The static combination of states in an ontology translates directly to the combination of the processes which apply to these states and changes them. It requires for the execution of a second order function to compute the next state.

Two combination methods are identified and described here: products of states, which lead to a sum of affordances, and indexed set of states, which lead to products of index and operations. Theoretical considerations [2] suggest that other combinations can be constructed from these two. Generic definitions, which translate from static to dynamic with the function *next* for these are given. From the static description of the combination of states the dynamic process definition follows automatically; executable programs result without further action.

The paper is structured as follows: it clarifies terminology and introduces the example, which is given as english text and (partially) formalized. It then justifies the approach and gives the specifications ADT style for the two base mechanism used. Section 8 and 9 introduces the two combination methods and section 10 shows how they are applied. Section 11 indicates how two simplifications assumed here can be lifted.

2 Ontological Commitments and Terminology

I assume here for a physical model (i.e mostly a "physical objects and processes" ontology, i.e. what I called tier 2 of a tiered ontology):

1. There is one physical reality, which has a state.
2. Processes change the state.
3. Objects have an identity and a state.
4. The evolution of reality in time are changes in state.

5. *Processes* change the *state* of *objects* but the object maintains its *identity*.

For simplicity, it is assumed (1) that processes are instantaneous and change the state of the object immediately; there is no gradual change (more like social process: one does not get gradually married),

(2) The article focuses on dynamic models of objects and does not discuss models of continuous space-time and physical processes there (i.e. tier 1 of the tiered ontology); for combinations there see [9].

(3) the objects share no states, thus the object state can replace the object identity.

(4) the processes applied do not violate restrictions; to keep the example simple, no checking is included.

It will be shown how the simplifications (3) and (4) can be lifted in section 11 and 12.

The objects are modeled as functions from $ID \rightarrow worldstate \rightarrow objectState$ and process as a function $state \rightarrow state$, possibly with additional parameters, to give specifics of the process. Processes form a category: there is an identity, processes compose and composition is associative:

$$\begin{aligned} identity: &\quad id \cdot a = a \\ composition: &\quad \forall a, \forall b, \exists c\ c = a \cdot b \\ associativity: &\quad a \cdot (b \cdot c) = (a \cdot b) \cdot c \end{aligned}$$

3 Example: Planning to get a Passport

3.1 Description

In the example an agent wants to apply for a new passport and therefore has to meet two main requirements : (1) being at an office withing the opening hours and (2) providing certain objects (e.g.: the old Passport, a photograph, some money). In order to meet these conditions the agent comes up with a plan by consulting a mental map that provides information about where to get objects and what are the pre-conditions for the acquisition of them. (For a more detailed description of the example see also [1]). The task hence exhibits spatio-temporal as well as equipment conditions, although in this paper we ignore temporal factors. Based on the locations and pre-conditions of the objects the agent determines a plan that consists of an ordered set of actions that lead to the desired state. These actions involve the movement from home to an ATM to withdraw money and consequently to a shop where a photograph is taken.

3.2 Ontology

The ontology consists of objects with a state, i.e. agent and stores, and objects without a state: Location, Equipment. The Agent can move and pick up equipment, thus has states for Location and a Container for Equipment. The

stores have a policy, e.g. to obtain a passport from the office, one has to deliver Money, Photo and old Passport, and contain equipment (i.e.: a set of objects). The stores are at locations and interaction between an agent and a store is restricted to the store at the location where the agent currently is. There are links between the locations where the agent can move.

The example invokes two mechanism: Container and Location, with the related processes to change the respective states.

3.3 Formalized Ontology

Modeling the example in protege[1] using OWL [14] results in a static description of the objects and concepts that play a role in the task (figure 1). In the model we defined an *Action-class* that contains different types of actions such as *Container-operations* (i.e.: add,drop), *Move-operations* and *Shop-operations* (i.e.: exchange). We assume that these are the minimum operations necessary to complete the given task. The other classes describe the above mentioned non-state objects and state-objects. An agent is defined by some equipment and some actions. A store is defined by the equipment it contains and the policy it imposes. A policy itself translates into a set of actions that are possible at the store. While a *FreeType* offers the Container-operations *add* and *drop*, the *ShopType* store only offers *exchange*. The shortcomings of OWL become apparent in the representation of the action ordering. For a system to plan the task, some representation of sequences are necessary, to be able to evaluate the combination of actions in relation to the desired outcome. Thus what we cannot do, is modelling processes without utilizing solutions to the frame-problem [13]. These do often resemble or reconstruct some of the combinators of lambda-calculus (e.g.[12]), what builds the basis for the work presented here.

[1] http://protege.stanford.edu/

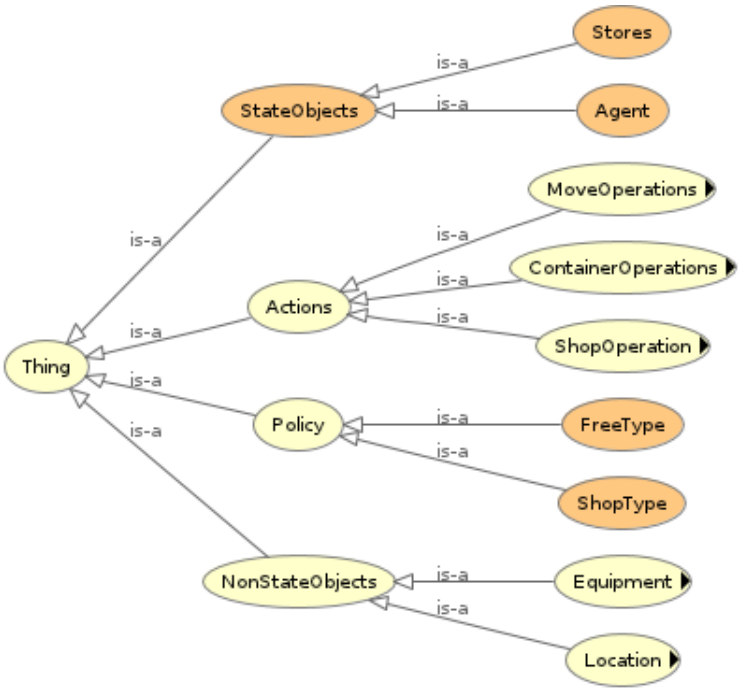

Figure 1: The asserted hierarchy of the passport example modeled in OWL.

4 Semantics: Mechanism are collections of related processes

Mechanisms are conceptually related processes - filling and emptying a bottle, driving from one place to another and then to the next. Such related processes - independent of the particular application's terminology are (small) theories; Casati used the term "theoritas" to point out that these theories are so small and commonplace that we do not pay attention to them [3]. They are closely related to Image Schemata, discussed in cognitive science [11].

Mechanism are related processes, which act on the same state; e.g., opening and closing a door. A mechanism is a set of states (a domain) and some related functions which change the state (also called operations or process). For discrete systems, e.g. traveling in a car, the world can be modeled as a set of links between locations; the state of an object is its location and moves are transitions from one location to another one.

5 Complex Behaviour is the result of combination of simpler mechanism

The *complexity assumption* is that the complexity we observe in the world is the result of the interaction between a very large number of small mechanisms, but each mechanism itself is simple. It is a philosophical question whether the mechanisms are simple or just conceived as simple, because humans can only understand and construct simple base mechanism. A complex mechanism consists of an interacting combination of things and produces complex, observable, behaviors from the simple behaviors of the constituent mechanisms.

6 Base Mechanism

The program in this article is to construct bottom up, we therefore start with two simple mechanism: Location and Container.

6.1 Container

The container is generic for all sorts of things. The axioms are given:

```
is_empty (empty) = True
is_empty (put a) = False
remove a ((put b) s) = if a==b then s
      else put b (remove a s)
remove a (empty) = error
contains a ((put b) s) = if a==b then True
      else contains a s
contains a (empty) = False
```

6.2 Location

Locations are represented as names, additional information could be attached with functions Loc -> Value, but is not required for this example (or is done as Loc -> Store)

The axiom is simple: the agent is at the location where it last moved to.

```
is_at a (moveTo b) = a == b
```

7 Representation of Actions

The "order" for executing a process is explicitly represented

```
data ContainerOp = Add Equipment | Drop Equipment
data MovementOp = Movement Loc Loc
```

The execution of these actions is

```
next  (Add a)  =  (put a)
next (Drop a) =  (remove a)
next (Movement a b)  =  (moveTo b)
```

8 Combination of Processes as Product-Sum

An Agent has a location and has a container with the things carried around. This is modeled as what we call a product-sum-combination. The state of the agent is the *product* of the individual states for location and the container; the processes applicable are the *sum* of the processes applicable to container and location [6].

The agent is declared as a pair of location and container with functions to access the constituting parts and functions to operate on them as the sum of the functions applicable to either part.

```
type AgentState = (Loc, Container)
type AgentOp = Either MovementOp ContainerOp
```

The Agent (i.e. the tuple) uses the function next applied to the first or second part (i.e. the location or the container). This is defined generically and applies automatically to all product-sum combinations:

```
next  = either (first . next) (second . next)
```

9 Indexed sets of States

The world of this example has stores (or store-like mechanism) at different locations and is represented as an array of (store) states, indexed by location. The operations are a product of the location and the operation applicable there. The *next* function uses the given process and *adjusts* the appropriate location.

```
type StoreState = (ShopType, Container)
type StoreOp = Either ShopOp ContainerOp

type MapOfStore = Map Loc StoreState
type MapOfStoreOp = (Loc, StoreOp)

next (l, op) = adjust (next op) l
```

10 Execution follows from a dynamic description

The ontology is translated into the structure of states above given for Agent, Store and MapOfStore. Adding Top-States and Top-Processes

```
type TopState = (AgentState, MapOfStore)
type TopOps = Either AgentOp MapOfStoreOp
```

as well as an initial state is all we need to have a dynamic description that can be executed. The functions moveop, pickupFrom, add2agent and pickUp are just to help with writing operations applied to a state to compute a next state:

```
emptyEquipmentContainer = empty :: Container
startAgent = (Home, emptyEquipmentContainer)

moveOp a b = next(Left . Left $ Movement a b ::TopOps)
pickUpFrom at as = Right (at, Right (Drop as))::TopOps
add2agent as = Left (Right (Add as))::TopOps
pickUp at as = next(pickupFrom at as) .
               next(add2agent as)
```

A state in which the Agent has money and is at the Shop can be computed by:

```
state3 = (pickUp Home Money).(moveOp Home Shop)$state0
```

11 Further Extensions

The representation of objects here as states assumes that no states are shared between objects. This is not usually the case. For example, a door is open or closed, and this state is relevant for both adjoining rooms and can be changed from either room. Such situations are handled by constructing the world state in a database, where individual object states are obtained by a functions from ID to state. Then each state is represented exactly once, but accessible from multiple objects.

The example above does not include any checks - movements are not restricted by a street graph, equipment can be picked up, even when the agent is not present at the store etc. etc. A common approach is to include appropriate checks and to wrap the state of the world into a datatype which signals either a valid state or a constant (in Haskell [10] the *Maybe* data type is typically used). It is obvious, that much of real world specifications is expressed in terms of restrictions on the execution. For example, above a rule like "every equipment is always in a container" is enforced if $Add(x)$ and $Drop(x)$ are always used in conjunction and shop policies must translate in *Add* and *Drop* processes etc.

The specification given here is (nearly) sufficient for planning optimal sequences of processes. In addition to the function *next* needs only two functions $affords$, i.e. what are possible actions given the current state, and *cost*, i.e. what is the "cost" expressed in some numeric value of executing the given process starting with the current state. With the three functions *next*, $affords$ and *cost*, a shortest path algorithm as given by [5] can be run, without constructing the state-transition graph explicitly.

12 Conclusion

The contribution uses the well known method for specification of simple mechanism using abstract data types and combines them to achieve complex specifications for objects and processes. The static description of objects, consisting of states, is sufficient to construct processes that act on the appropriate states. Two methods for combining mechanism to objects are identified and it is shown, that the application of processes to these combinations of states can be described generically, i.e. from a description of the structure of the objects consisting of states (which are typed to connect to specific mechanism) a dynamic, executable specification of the processes is obtained automatically. The approach can even be used to plan optimal sequences of processes to achieve desired future states.

References

[1] A. Abdalla and A. U. Frank. Towards the spatialization of pim-tools. In *GIZeitgeist 2012: Proceedings of the Young Researchers Forum on Geographic Information Science*, Muenster, March 2012. ifgiPrints.

[2] Richard Bird and Oege de Moor. *Prentice Hall International Series in Computer Science*, chapter Algebra of Programming. Prentice Hall Europe, 1997. copy AF Inv.No. 602.040 and 6023.308 v. 29.6.99.

[3] R. Casati. The structure of shadows. *Life and motion of socio-economic units*, 8:88, 2001.

[4] Nicolas Chrisman. Aag tobler lecture. 2012.

[5] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, (1):269–271, 1959.

[6] A. Frank. Wayfinding for public transportation users as navigation in a product of graphs. *VGI (Spezialband)*, 2:195–200, 2007.

[7] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. *Initial Algebra Semantics and Continuous Algebras.* copy, 1977.

[8] J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10(1):27–52, 1978. Copies.

[9] B. Hofer and A. Frank. Composing models of geographic physical processes. *Spatial Information Theory*, pages 421–435, 2009.

[10] S.P. Jones. *Haskell 98 language and libraries: the revised report.* Cambridge Univ Pr, 2003.

[11] G. Lakoff and M. Johnson. *Metaphors We Live By.* University of Chicago Press, 1980. Source: Max Egenhofer.

[12] H.J. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R.B. Scherl. Golog: A logic programming language for dynamic domains. *The Journal of Logic Programming*, 31(1-3):59–83, 1997.

[13] John McCarthy and Patrick J. Hayes. *Machine Intelligence 4*, chapter Some Philosophical Problems from the Standpoint of Artificial Intelligence, pages 463–502. Edinburgh University Press, 1969. reprint in book 'Formalizing Common Sense', edited by V. Lifschitz (copy AF, Inv.No. 601.906).

[14] D.L. McGuinness, F. Van Harmelen, et al. Owl web ontology language overview. *W3C recommendation*, 10:2004–03, 2004.