Using Reflective Logic Programming to Describe Domain Knowledge as an Aspect*

Maja D'Hondt[1], Wolfgang De Meuter[2], and Roel Wuyts[2]

[1] System and Software Engineering Laboratory
[2] Programming Technology Laboratory
Brussels Free University, Pleinlaan 2, 1050 Brussels, Belgium,
`mjdhondt | wdmeuter | rwuyts@vub.ac.be`

Abstract. Software applications, mostly consisting of an algorithm applied to domain knowledge, are hard to maintain and to reuse as a result of their hard coded combination. We propose to follow the principles of aspect-oriented programming, separating the domain from the algorithm and describing them in a logic and conventional programming language respectively. In this paper, we report on an experiment that was conducted to validate this hypothesis, and to investigate the requirements of a programming environment for this configuration. An already existing environment that uses a logic meta-language to reason about object-oriented systems, SOUL, is used as a starting point for this experiment. The result is a working implementation in SOUL, which validates our ideas, reveals mechanisms that require more research, and points to other features that should be included.

1 Introduction

In [1] we argue that separating the *domain* from the *algorithm* in software applications, would solve a lot of maintainance and reuse problems. Applying *aspect-oriented programming* or short *AOP* [2] to model the domain as an aspect program and the algorithm as the base program, allows them to evolve independently from one another. Note that in this paper, by algorithm we mean the functionality of a software application or in general any program, whereas a domain or domain knowledge denotes concepts and constraints that model the real world and which an algorithm can be applied to [1].

Hence, following the AOP paradigm, we propose to express domain knowledge in an appropriate environment acting as an aspect language. When investigating techniques available in (amongst others) the artificial intelligence community for representing knowledge, we come across a number of languages and formalisms based on first-order predicate logic. The algorithm, on the other

* This work is partly sponsored by the *Vlaams Instituut voor de Bevordering van het Wetenschappelijk-Technologisch Onderzoek in de Industrie.*

[1] Note that, in some cases, an algorithm can also be viewed as being part of domain knowledge.

hand, is the base program and can be implemented in any conventional programming language such as C, Java and Smalltalk.

In order to conduct initial experiments investigating our hypothesis that software applications would benefit from separating the domain and the algorithm, we require a programming environment that is a symbiosis between a domain aspect language and a conventional programming language. A suitable candidate is the *Smalltalk Open Unification Language* or *SOUL* [3] [4], a declarative framework that uses Prolog to reason about the structure of Smalltalk programs. SOUL was originally developed as a validation for the use of logic meta-languages for expressing relationships in object-oriented systems. In this paper we take advantage of this construction to represent domain knowledge in the Prolog layer and algorithms in Smalltalk. Therefore, Prolog is no longer used as meta-language for reasoning about Smalltalk programs, rather it serves as an aspect language for describing aspect programs on the same level as the base programs in Smalltalk.

In the rest of this paper, we describe an experiment involving an example inspired by a real-world geographic information system (GIS). Although not originally designed for this purpose, we use SOUL and Smalltalk to turn our example into a working aspect-oriented program, thereby providing proof of concept of our ideas. In addition to this, the experiment reveals features that should be incorporated in a programming environment for AOP with domain knowledge as an aspect. Some of these features are not yet implemented in SOUL, but present a mere technical issue. Other features, however, require more investigation.

In the remainder of this text, the *slanted type style* is used to denote concepts from the domain knowledge, whereas algorithms are written in `typewriter type style`.

2 An Example

A GIS, for example a car navigation system, is applied to a vast amount of geographic data, modelled as a planar graph. The nodes, edges and areas of this graph have properties attached to them such as a hotel, a street name or an industrial area respectively. These geographic data are actually the domain knowledge of a GIS. Our example concerns *roads*, *cities* and *prohibited manoeuvres* (Fig. 1). The latter is modelled on two cities, signifying that it is prohibited to take the road from the first city to the last.

To this sub domain a shortest path algorithm is applied, more specifically the *branch and bound algorithm* below:

```
branchAndBoundFrom: start to: stop
   |bound|
   bound := 999999999.
   self traverseBlock: [:city :sum|
   city free ifTrue: [sum < bound ifTrue: [city = stop
```

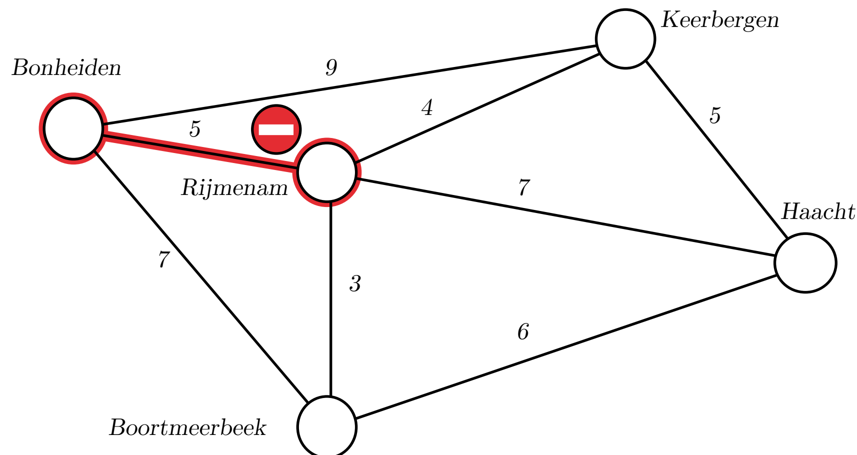

Fig. 1. A planar graph representation of the domain knowledge. The circles denote cities, whereas the lines indicate roads that connect two cities. To each road the distance between the two connected cities is attached. A prohibited manoeuvre is shown from the *city Rijmenam* to the *city Bonheiden*.

```
      ifTrue:  [bound := sum ]
      ifFalse: [self branch: city sum: sum]]]].
   self traverseBlock value: start value: 0.
   ^bound

branch: node sum: sum
   city free: false.
   city roads do: [:road|
   (self isProhibiedFrom: city by: road) ifFalse:
      [self traverseBlock value: road next
                          value: sum + road distance].
   city free: true.
```

By way of introducing the example, we presented it programmed in a conventional way: the domain knowledge and the algorithm are both implemented in Smalltalk. Note that the algorithm implicitly selects the shortest road first, because the instance variable `roads` of a `city` is a sorted collection. Figure 2 shows a UML diagram of the domain and the algorithm.

This branch and bound program illustrates that the domain and the algorithm cannot evolve independently from each other: the domain knowledge concerning prohibited manoeuvres cross-cuts the implementation of the algorithm.

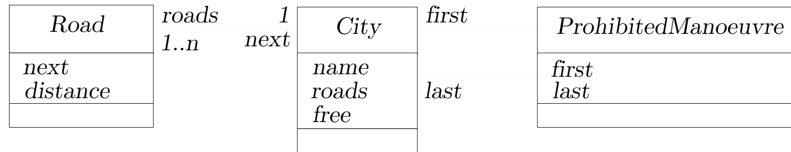

Fig. 2. An UML diagram of the branch and bound algorithm. We assume that accessors and mutators are implemented for the instance variables, which are used in the branch and bound program.

3 A First Experiment

The example, which was described in the previous section as a single Smalltalk program, is now transformed into an aspect-oriented program in SOUL. Using the Prolog layer to describe the domain knowledge, we get the following result:

Fact city (Rijmenam)
Fact city (Boortmeerbeek)
...
Fact road (city (Rijmenam), city (Boortmeerbeek), [3])
Fact road (city (Keerbergen), city (Rijmenam), [4])
...
Fact prohibitedManoeuvre (city (Rijmenam), city (Bonheiden))
Rule roads (?current, ?newResult) if
findall (road (?current, ?next, ?distance),
road (?current, ?next, ?distance), ?result)
privateRoads (?current, ?result, ?newResult).
Rule privateRoads (?current, ?result, ?newResult) if
prohibitedManoeuvre (?current, ?next),
removeRoad (?result, road (?current, ?next, ?distance), ?newResult)
Fact privateRoads (?current, ?result, ?result)

The base program in Smalltalk looks like this:

```
branchAndBoundFrom: start to: stop
   |bound|
   bound := 999999999.
   self traverseBlock: [:node :sum|
   node free ifTrue: [sum < bound ifTrue: [node = stop
      ifTrue:  [bound := sum ]
      ifFalse: [self branch: node sum: sum]]]].
   self traverseBlock value: start value: 0.
   ^bound

branch: node sum: sum
   node free: false.
```

```
node edges do: [:edge|
self traverseBlock value: edge next value: sum + edge distance].
node free: true.
```

Note that it is basically the same as the first version of the algorithm presented in the previous section. We have changed the variables and methods `city` to `node` and `road(s)` to `edge(s)`, to stress that domain knowledge concerning cities and roads is no longer part of the algorithm. There are still nodes and edges in this otherwise domain independent algorithm, because a branch and bound algorithm is graph-based and thus requires the use of these concepts. This imposes a constraint on the domain this algorithm can be applied to, but also indicates the *join points* between the two: *cities* map to `nodes` and *roads* map to `edges`.

Another difference with the previous conventional implementation, is that the base program is no longer responsible for the selection of the next roads to be visited: in the domain layer, the rule *roads* unifies with *?newResult* a list of next roads omitting prohibited manoeuvres. This delegation to the Prolog layer is possible, because Prolog queries can be explicitly launched from Smalltalk code. Thus, when the message `edges` is sent to the variable `node` in the base program, this is translated to a query, for example:

q Query roads (city (Rijmenam), ?newResult)[2]

This brings us to another point: the value at run-time of the variables `node` and `edge` in the base program. The algorithm works with placeholder objects, instances of the classes `Node` and `Edge`, which do nothing more than wrap corresponding Prolog facts, *city* and *road* respectively, thus transforming them to Smalltalk objects. Accessing these objects is delegated to the domain layer as a query (see previous example), since these objects do not actually store any values other than a Prolog fact. The mechanism of manipulating Prolog objects in Smalltalk and vice versa, necessary for fixing the join points of the aspect-oriented program, is referred to as the *linguistic symbiosis* of those two languages [5]. The idea is the following:

$$\mathrm{up}(\mathtt{s}) = p \qquad \text{if } \mathtt{s} = \mathrm{down}(p)$$
$$= \mathit{smalltalkFact}(\mathtt{s})$$

$$\mathrm{down}(p) = \mathtt{s} \qquad \text{if } p = \mathrm{up}(\mathtt{s})$$
$$= \mathtt{Prologobject\ new\ with:}\ p$$

where Prolog is the up layer and Smalltalk is the down layer, and where `s` is a Smalltalk object and p is a Prolog fact. *Upping* a *downed* Prolog object returns the original unwrapped Prolog object. *Upping* a Smalltalk object means wrapping it in a Prolog fact *smalltalkFact*. Vice versa, *downing* an *upped* Smalltalk

[2] The prefix *q Query* is required by SOUL syntax.

object returns the Smalltalk object whereas *downing* a Prolog fact wraps it in an instance of the `PrologObject` class from Smalltalk. When Smalltalk sends a message to a Prolog fact, for example to retrieve edges (roads) from *city(Haacht)*, the fact is implicitly *downed* and the result is *upped* again [6]. This linguistic symbiosis is not yet implemented in SOUL and is therefore explicitly and not very elegantly programmed in this experiment, which nevertheless provides a proof of concept. Further work certainly includes embedding this mechanism in SOUL, thus making it implicit and invisible to the programmer.

However, the linguistic symbiosis to enable AOP with the domain aspect requires the *downed* Prolog facts to have a state, as the messages `free` and `free:` in the base program indicate. These messages are used to let the algorithm know if a node has already been visited in order to avoid cycles. This property attached to nodes is purely algorithmic and should not be incorporated in the domain knowledge about cities. Therefore, when a Prolog fact is *downed* for the first time, the programming environment should attach an instance variable `free` and corresponding accessor and mutator to the instance of the class `PrologObject` in Smalltalk that wraps it. This wrapper should be recycled when the same Prolog fact is *downed* again, so that the value of `free`, in other words the state of the wrapper, is not lost.

4 A Second Experiment

For the next experiment, we strip the domain knowledge of prohibited manoeuvres and add a new concept: the *priority manoeuvre* as explained in figure 3.

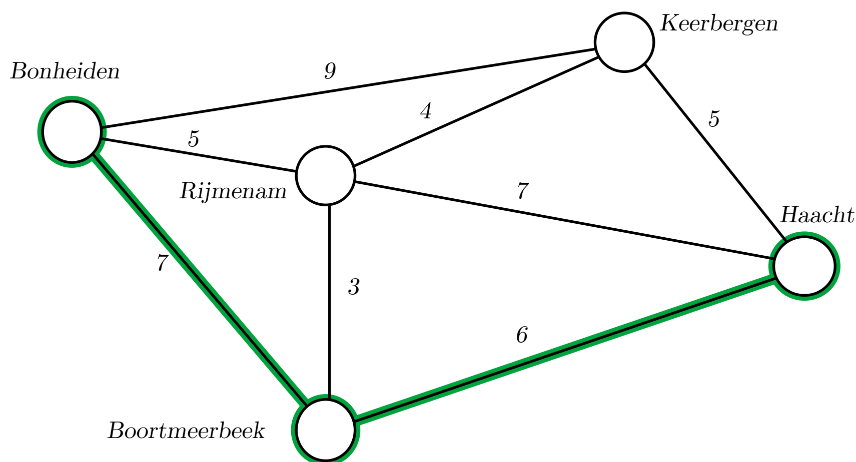

Fig. 3. The priority manoeuvre that concerns the cities *Haacht*, *Boortmeerbeek* and *Bonheiden*. When in *Boortmeerbeek* and coming from *Haacht*, the road to *Bonheiden* is preferred over any other road, even when that road is shorter.

The domain knowledge is extended and replaced with the following:

Fact priorityManoeuvre (city(Haacht), city(Boortmeerbeek), city(Bonheiden))
Rule roads (?current, ?previous, ?newResult) if
    findall (road (?current, ?next, ?distance),
    road (?current, ?next, ?distance), ?result),
    privateRoads (?current, ?previous, ?result, ?newResult)
Rule privateRoads (?current, ?previous, ?result,<road (?current, ?next, ?distance) | ?newResult>) if
    priorityManoeuvre (?previous, ?current, ?next),
    removeRoad (?result, road (?current, ?next, ?distance), ?newResult)
Fact privateRoads (?current, ?previous, ?result, ?result)

If we would implement the branch and bound algorithm applied to this domain in a conventional program, the algorithm would require the passing of an extra parameter `previous` in order to find priority manoeuvres, as shown below:

```
branchAndBoundFrom: start to: stop
   ...
   self traverseBlock: [:previous :node :sum |
      ...
      ifFalse: [self branchFrom: previous to: node sum: sum]]]].
   self traverseBlock value: nil value: start value: 0.
   ^bound

branchFrom: previous to: node sum: sum
   ...
```

As with the `free` instance variable in the previous section, the domain knowledge cannot be burdened with this algorithmic information. But neither should the algorithm be tangled with an extra parameter that is needed to test a domain constraint concerning priority manoeuvres. In our working implementation, we use the domain knowledge described above and the same algorithm as was presented in the previous section. This shows that the domain can evolve independently from the algorithm. In order to achieve this, we created an object that runs in background in parallel with the algorithm, secretly remembering the previously visited node. Although not very elegant, our temporary solution again hints at what kind of mechanism should be incorporated in an environment for AOP with domain knowledge as en aspect.

5 Conclusion

This paper reports on an experiment on aspect-oriented programming with domain knowledge as an aspect, using a reflective logic language as the aspect language. On the one hand this experiment was conducted to show that the domain and the algorithm of a program can evolve independently from one another

when using the AOP paradigm and regarding domain knowledge as an aspect. We validated this hypothesis on a small scale, since the experiment involved limited domain knowledge and a relatively simple algorithm. On the other hand, despite this small scale, the experiment immediately revealed some properties that a programming environment which supports this programming style should have. First of all, it should incorporate a language based on first-order predicate logic for representing the aspect, such as Prolog, and a more conventional programming language for the algorithm. Moreover, the programming environment should provide a language symbiosis between the aspect language and the base language, to allow transparent manipulation of objects from one language in the other. This mechanism is required in order to express the join points between objects in the aspect language (up) and objects in the base language (down). In addition to this, the algorithm should be able to attach algorithmic properties to domain knowledge objects. More precisely, the base program should have the possibility to add instance variables and methods to a *downed* domain object. Finally, this experiment showed that some kind of memory manager should be conceived that runs in the background and that passes otherwise lost information to either the algorithm or the domain layer.

References

1. D'Hondt, M., D'Hondt, T.: Is domain knowledge an aspect? ECOOP99, Aspect-Oriented Programming Workshop (1999)
2. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J.: Aspect-oriented programming. In Proceedings of ECOOP (1997)
3. Wuyts, R.: Declarative reasoning about the structure of object-oriented systems. In Proceedings of TOOLS USA (1998)
4. Wuyts, R.: Declaratively codifying software architectures using virtual software classifications. Submitted to TOOLS Europe (1999)
5. Steyaert, S.: Open design of object-oriented languages, a foundation for specialisable reflective language frameworks. PhD thesis, Brussels Free University (1994)
6. De Meuter, W.: The story of the simplest MOP in the world, or, the scheme of object-orientation. In Prototype-Based Programming (1998)