Can Domain-Specific Languages Benefit from Linguistic Symbiosis?

Kris Gybels*
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Elsene, Belgium
kris.gybels@vub.ac.be

July 5, 2003

Abstract

When considering domain-specific languages that are not purely static, the question arises how programs written in the DSL can interact with those of the base language. We propose to apply the concept of *linguistic symbiosis* as a standard for evaluating mechanisms that allow such an interaction. Linguistic symbiosis in general refers to the ability of programs written in different languages to interact transparently as if they were written in the same language.

1 Introduction

Linguistic symbiosis is a concept that arose from work on reflectively extensible interpreters. In the work of Ichisugi et al. [7] and Steyaert [9] and De Meuter [8], linguistic symbiosis refers to the ability of base-level objects and meta objects to send each other messages when the base level and meta level languages are not the same. Reflection can then be achieved by allowing inter-level "travelling" of objects. When a meta object travels to the base level it can be sent the same messages by base objects as can be sent by the other meta objects that form the interpreter. This allows basic introspection and intercession of the interpreter. If a base-level object supports the message protocol expected by the interpreter it can take the place of a meta object, which then allows changing the rules of interpretation themselves.

While linguistic symbiosis as a means for reflection is certainly interesting to the REPLS workshop, it is a generalization to base level interaction which we wish to discuss here. We recently explored the application of linguistic symbiosis to multi-paradigm programming where base-level parts

*Research assistant of the Fund for Scientific Research - Flanders (Belgium) (F.W.O.)

of a single program need to interact [2]. Two particular cases we explored were the implementation of business rules [4], a concept from the domain of business software, and the implementation of component configuration rules for generative programming [5]. In both cases we made use of logic programming to implement these "domains". In the remainder of the paper we present these cases and discuss how linguistic symbiosis is exploited therein.

2 Business Rules

When developing software to support a business, one needs to distinguish between the business model and business knowledge governing the global policies applied by the business. The former is normally implemented in object-oriented programming, while the latter is implemented as a set of business rules, usually in some suitable language based on logic programming.

Because two different languages are used to express two parts of the the same application in, the issue arises how the two are connected. Figure 1 illustrates the general connection problem: one the one hand we have objects modeling the store and its customers, on the other rules stating when customers are allowed discounts. At certain points the rules need to access information that is stored in the objects, and objects need to be informed of "decisions" taken by the rules.

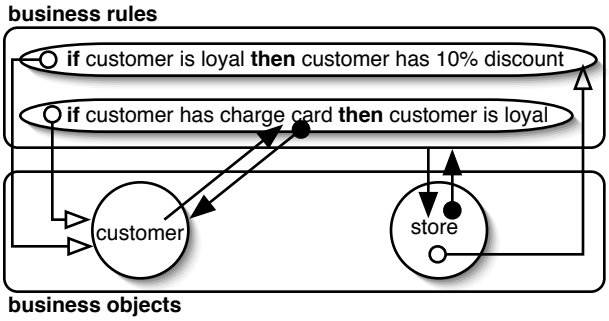

Figure 1: Connecting business objects in the core application functionality and business rules.

We found that in existing business rule systems several approaches are used to deal with the connection problem [4]. In some an explicit mechanism for switching between the OO and BR language is used, in others programmers need to explicitly specify how the concepts from the one (objects) can be mapped to the one from the other (relations), some use an implicit mapping etc.

We proposed to use the concept of linguistic symbiosis as a standard to evaluate the different approaches by. The more transparent inter-calling of

```
member(?x, <?x | ?rest>).
member(?x, <?y | ?rest>) if
  member(?x, ?rest).

<?x | ?rest> contains: ?x.
<?y | ?rest> contains: ?x if
  ?rest contains: ?x.
```

Figure 2: Comparison of list-containment predicate in classic and new SOUL syntax.

program elements in different languages is, the less programs are cluttered with explicit switches. Another benefit of symbiosis would be to allow for easy business rule refactoring: when a business evolves some methods may become easier to express in the BR language because their behavior involves taking into account several business rules. We would then like to replace the method with rules, without having to change the code that invoked that method.

3 Symbiotic Languages

To experiment with base-level symbiosis for business rules we used the SOUL [10] logic language. SOUL is implemented in Smalltalk and originally was based on Prolog with some minor syntactical differences and an additional mechanism for manipulating Smalltalk objects from SOUL. This allowed for a simple interaction approach where SOUL could escape to Smalltalk for sending the objects messages. Invoking queries from Smalltalk involved explicit construction of a SOUL evaluator object and sending it messages to execute queries and retrieve results. Thus the interaction either way was not very transparent.

To allow for better symbiosis we made some adaptations to SOUL [6]. A key change was to change the syntax from Prolog-like syntax to one that is more like the message sending syntax of Smalltalk. Figure 2 contrasts the classical `member` predicate of Prolog with the new `contains:` version in SOUL. This syntactic likeness makes it easier to define a semantic interaction through mapping of objects/messages and relations/predicates. We changed the evaluation process of both Smalltalk and SOUL as follows: when SOUL does not find a rule for a predicate, the predicate will be translated to a message. When Smalltalk does not find a method for a message, the message is translated to a query. The translation itself is rather straightforward, though there are some issues in how to deal with multiple results from queries etc. Our point here is mostly to illustrate how symbiosis is used, so we will not discuss the details of its working and origin, we refer to our paper “SOUL and Smalltalk - Just Married” [6] for further details.

4 Business Rules in SOUL

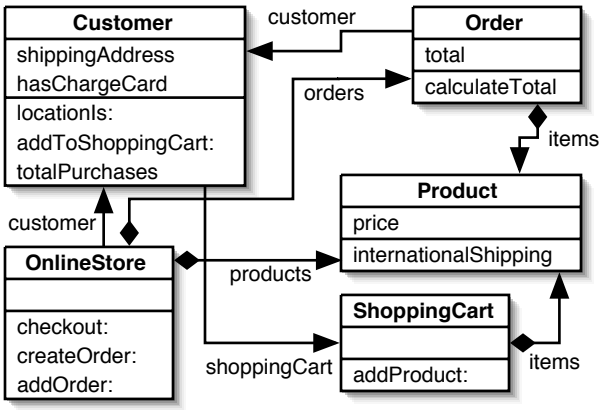

Figure 3: A class diagram of the online store

Figure 3 shows the class diagram for a simple business application, an online store, which we will use to illustrate how the symbiosis of SOUL and Smalltalk is used to actually implement business rules.

A typical task for an online store is to calculate the total price of all items a customer bought and apply any discounts. This is implemented by the method `calculateTotal` in the class `Order`:

```
calculateTotal
 total:=0.
 self items do:[:i|total:=total+i price].
 total:=total-(total*customer discount/100)
```

Whom to give a discount is an example of business rule knowledge. Some simple rules we can implement for this in SOUL are:

```
?c discount = 10 if ?c premierCustomer
?c discount = 5 if ?c loyalCustomer
?c discount = 0 if ?c ordinaryCustomer

?c premierCustomer if
 ?c totalValueOfPurchases = ?p &
 ?p greater: 10000
```

These rules simply express that a customer is given a 10% or 5% discount when he is a premier customer or a loyal customer respectively. When a customer falls into these categories is again governed by business rules, one for "premier customer" is also shown above.

In this particular example there are two points where a language switch occurs. In the `calculatePrice` method the message discount is sent to a customer object. There is actually no method for this message, so

```
?featureSet disallowedFor: ?account if
  ?featureSet containsFeature: credit &
  ?account disallowedCreditRating.

?account disallowedCreditRating if
  ?account balanceHistory = ?history &
  ?history badHistory.

?account disallowedCreditRating if
  ?account owner = ?owner &
  ?owner age = ?age &
  ?age isBelow: 22.
```

Figure 4: Example rules expressing dynamic disallowed feature set inclusion constraints.

because of our changes to Smalltalk it will be translated to the query `?customer discount = ?d` with `?customer` holding the customer object to which the message was sent. The query's solution for the variable `?d` will be returned as result of the message. Vice-versa a switch from SOUL to Smalltalk occurs in the rule for `premierCustomer` where the predicate `totalValueOfPurchases` is used. There is not actually a rule for this predicate, so it will be translated to a message to the customer object in `?c`.

5 Component Reconfiguration Rules

Another domain to which we applied the symbiosis concept in SOUL is that of component reconfiguration rules for dynamic generative programming. In a paper at last year's GPCE conference, Barbeau and Bordelau argued that a generator should become more of a configurator, able to (re)compose components at runtime [1]. In addition to checking the component inclusion constraints as documented in feature diagrams, such a configurator would also have to check certain dynamic constraints. Such as not giving a bank account a credit overdraw feature when its owner has a low income or is younger than 22.

We experimented with a simple Smalltalk component model for implementing the features in and checking whether the combinations are allowed. The feature components are implemented as prototype objects, which are easy to (re)compose in a delegation relationship. We used this to implement the bank account example from the GP book [3] with some additional feature composition rules implemented in SOUL.

Some of the rules are illustrated in figure 4. The `disallowedFor:` predicate is used by the configurator when a recomposition of components is requested, with the new feature set and the existing component object as arguments. The example `disallowedFor:` rule expresses that the feature set cannot contain the *credit* feature if the account is not allowed the feature. It is not allowed, as expressed in the rules for

`disallowedCreditRating`, when the account has a bad credit history or when its owner is younger than 22.

The difference between this approach and the one taken in the GP book is of course the use of the symbiotic logic language. Czarnecki and Eisenecker advocate the use of C++ metaprogramming for implementing static feature constraints, so presumably they would advocate the use of regular C++ for dynamic constraints. In both this case and the earlier business rule example we use a logic language because it is more adapted to expressing such rules, especially if more complex constraints which require some reasoning would need be implemented, while the symbiosis still allows it to interact easily with the object-oriented domain of the application.

6 Position

This paper's title reflects our position, or better question, for the REPLS workshop: can domain-specific languages benefit from linguistic symbiosis? Our experience with the concept is limited to the two cases presented here and so we are interested in learning about other domains where interaction between different languages could be useful. So far we can observe it is mostly relevant to dynamic domain specific languages. Most research into DSL seems to focus on providing domain-specific syntactic sugar wherein any concepts of the domain-specific language are translated to those of the host language as part of the compile-time desugaring process. In our approach we mapped concepts such as objects, messages, relations and rules at run-time. In fact, the merit of each language lies not in its *syntactic* fit to a particular domain, for we even changed the syntaxes to be more alike, but rather the different concepts and interpretation used.

References

[1] Michel Barbeau and Francis Bordeleau. A protocol stack development tool using generative programming. In D. Batory and C. Consel, editors, *Proceedings of Generative Programming and Component Engineering*, volume 2487 of *Lecture Notes in Computer Science*. Springer, 2002. 5

[2] Johan Brichau, Kris Gybels, and Roel Wuyts. Towards a linguistic symbiosis of an object-oriented and logic programming language. In Jörg Striegnitz, Kei Davis, and Yannis Smaragdakis, editors, *Proceedings of the Workshop on Multiparadigm Programming with Object-Oriented Languages*, 2002. 2

[3] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000. 5

[4] Maja D'Hondt and Kris Gybels. Linguistic symbiosis for the automatic connection of business rules and object-oriented application functionality. Submitted to conference on Automated Software Engineering 2003, 2003. 2

[5] Kris Gybels. Enforcing feature set correctness for dynamic reconfiguration with symbiotic logic programming. Position paper at the Eighth International Workshop on Component-Oriented Programming. 2

[6] Kris Gybels. Soul and smalltalk - just married: Evolution of the interaction between a logic and an object-oriented language towards symbiosis. Submitted to the Workshop on Declarative Programming in the Context of Object-Oriented Languages. 3

[7] Yuuji Ichisugi, Satoshi Matsuoka, and Akinori Yonezawa. Rbcl: a reflective object-oriented concurrent language without a runtime kernel. In *IMSA'92 International Workshop on Reflection and Meta-Level Architectures*, 1992. 1

[8] Wolfgang De Meuter. The story of the simplest mop in the world, or, the scheme of object-orientation. *Prototype-Based Programming (eds: James Noble, Antero Taivalsaari, and Ivan Moore)*, 1998. 1

[9] Patrick Steyaert. *Open Design of Object-Oriented Languages*. PhD thesis, Vrije Universiteit Brussel, 1994. 1

[10] Roel Wuyts. *A Logic Meta Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001. 3