Enforcing Feature Set Correctness for Dynamic Reconfiguration with Symbiotic Logic Programming

Kris Gybels
Programming Technology Lab
Vrije Universiteit Brussel
kris.gybels@vub.ac.be

April 14, 2003

1 Introduction

We propose here to enforce the correct configuration of generic components when dynamically reconfiguring them by encoding the rules governing the correctness using logic programming. Additionally, rather than using a classic logic language such as Prolog "as is", we want to use a logic language that allows the correctness rules to easily interact with the components. To achieve this we propose to use the concept of linguistic symbiosis and present a logic language that has such symbiosis with a component implementation language.

The motivation for this proposal lies in the work on Generative Programming (GP) in which generic components are delivered as a set of ready features which can be put together to form an actual component [3]. The actual component, usually simply a class, is composed by a generator which also checks that the selected features meet certain inclusion requirements documented in a feature diagram. As GP is usually used in a context in which components are generated statically these are about the only possible correctness checks to be performed.

As Barbeau and Bordelau pointed out in a paper at the last Generative Programming and Component Engineering conference, correctness checking gets more interesting when we start to consider dynamic reconfiguration of components [1]: "[...] what we often need to build in software engineering is a car that could reconfigure itself to move the driver seat from the left side of the car to the right side of the car when the car moves from France to England." When doing such reconfigurations additional constraints need to be taken into account: what country is the car currently in? Is it stopped before we start switching the driver seat? etc. They sketched a possible architecture for such a dynamically reconfigurable component in which the purely static GP generator is replaced with a dynamic configuration agent. Such an agent could be made to do reconfigurations on explicit request or by monitoring internal conditions of a component. Because such an agent would need to perform some form of reasoning to check the correctness of the new configuration we propose to write it in logic programming. Because it would need to access internal conditions

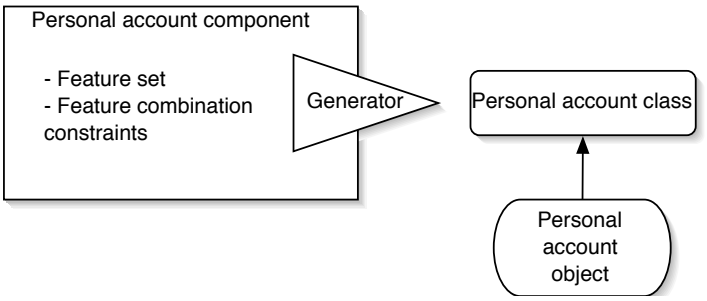

Figure 1: Illustration of the personal bank account as a GP component

of the component we propose the linguistic symbiosis mechanism to ease this interaction.

Instead of the car example, we will illustrate our approach with another example from the GP book [3]: personal bank accounts. We will briefly review this example in the next section. Then we digress from it for a moment and discuss linguistic symbiosis. After we show how it can be used in the implementation of the account example. We end the paper with a short discussion of questions on the wider applicability of this idea to Component-Oriented Programming and related work.

2 Personal Bank Accounts

The personal bank account component is illustrated in figure 1. As it is a component in the GP sense it consists of a set of parts implementing certain features that can be composed by its generator. The result of the generator is a class whose instances will support the selected features. Not all feature combinations are allowed and certain features force the inclusion of others. These constraints are normally documented in a feature diagram and enforced by the generator. The feature diagram for the example is shown in figure 2. Some of the constraints are that a bank account object always has the feature of being able to remember who its owner is, that it supports balancing but always one of four alternative forms of balancing etc.

The above depiction of GP is static in nature as the bank account component is used just once to generate a class. Figure 3 illustrates how we might think of GP in a more dynamic form. As illustrated we would like the ability to dynamically reconfigure the features of certain personal account objects to be of a different class output by the generator. In this sense we might think of the generator more as a (re)configurator.

Besides the feature inclusion constraints, a configurator may need to take constraints into account about the features' applicability, for example:

- When the credit rating of the account isn't high enough, the account shouldn't be given the "credit" feature.
- Before the introduction of the EURO, an account still expressed in the national currency could be changed to the EURO but not back again.

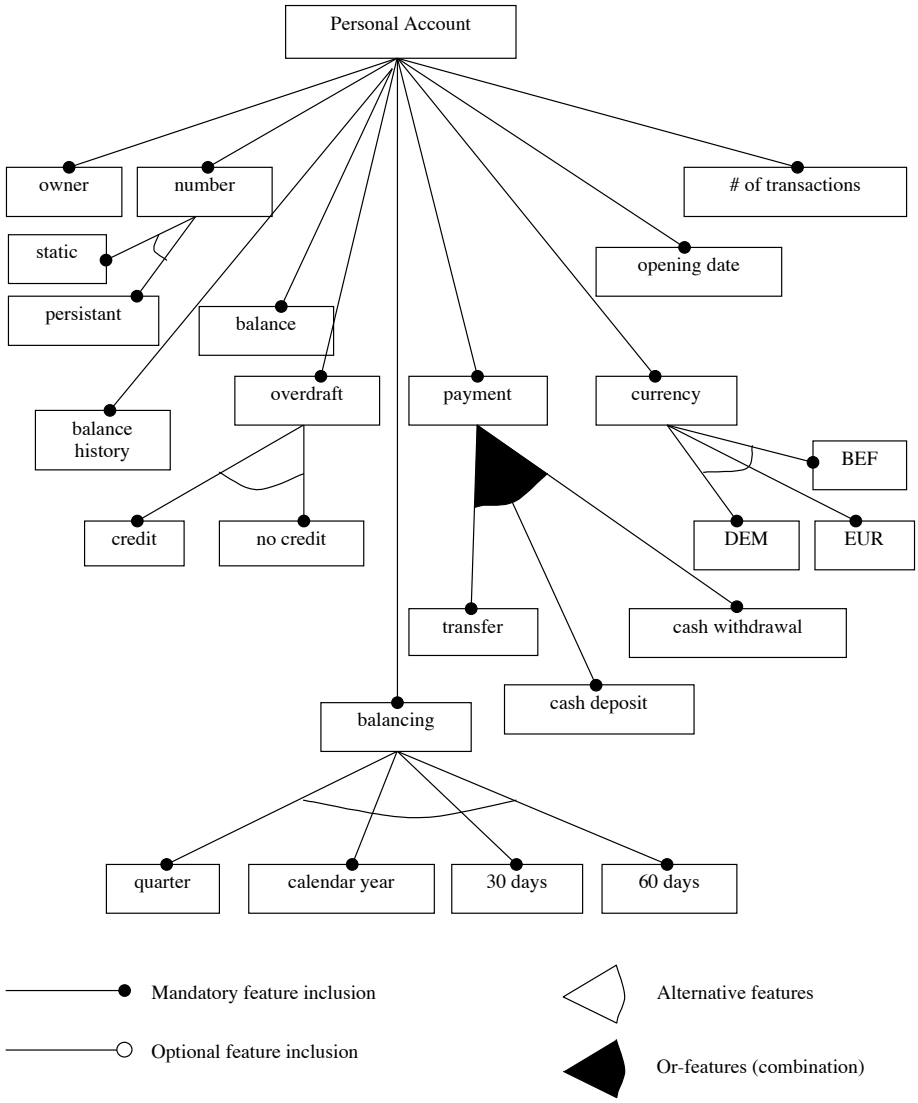

Figure 2: Feature diagram for bank account

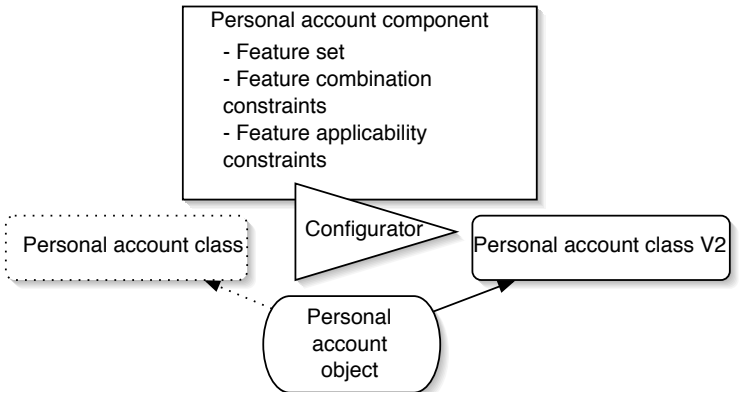

Figure 3: Illustration of a generator turning into a configurator

We will implement such constraints in the logic language we present next, focusing on the linguistic symbiosis with out chosen component implementation language, Smalltalk.

3 Linguistic Symbiosis

In general, linguistic symbiosis refers to the integration of two programming languages so that parts of a single program can be written in either language [5]. When the paradigms of the two languages are different, linguistic symbiosis can bring about multi-paradigm programming. This is the case here, as we want to combine an object-oriented language to write the components in and a logic language to write the composition process of the components in.

Linguistic symbiosis can range from a loose integration where program parts in either language can hop to the parts in the other one with a very explicit "escape" mechanism also allowing only the exchange of primitive data types to a very tight symbiosis where the hops are almost transparent [4]. The original version of the logic language we use, SOUL [10], was based on classical Prolog though it exhibited linguistic symbiosis by allowing objects to be bound to logic variables and messages could be sent to them by a special kind of term doing an "escape to Smalltalk". By incorporating this message sending as the basis for logic programming, rather than as such an "escape", we have achieved a better linguistic symbiosis of the two languages.

The changes in SOUL when compared to classic Prolog are thus the following:

Value system: variables can be bound to objects, not just Prolog values such as lists etc.

Syntax: Smalltalk message sending syntax is used for predicates.

Evaluation: instead of evaluating predicates by looking up a rule for them and recursively checking its conditions, the default is to simply send the message. When a rule exists for the message predicate however, it overrides this behavior to the normal Prolog behavior.

```
account includesAll: <balancing, transactionCounter,
                      openingDate, currency,
                      payment, overdraft, owner,
                      number, balance, balanceHistory>.
currency includesOneOf: <bef, eur, usd>.
payment includesCombinationOf: <transfer, cashDeposit,
                                cashWithdrawal>.
balancing includesOneOf: <quarter, calenderYear,
                          thirtyDays, sixtyDays>.
overdraft includesOneOf: <credit, noCredit>.
```

Figure 4: Facts corresponding to the feature diagram.

We will illustrate how this is used in the next section where we return to the account example.

4 Account Example Implementation

As we are concentrating on the configuration rules here, we will only briefly describe the other implementation aspects of composition in the account example. We have used a simple implementation in which the feature subcomponents of the account generator are implemented as objects. These objects are supposed to be composed using prototype-based delegation to form the larger account object. Though this is implemented in Smalltalk, it is done through an extension of the language and classes are not really needed. We have thus bypassed the explicit generation of a class as was depicted in figure 1 and let the generator generate objects directly. Because of the delegation this results in a less efficient implementation than would be possible but it is one in which the features supported by an object are easier to manipulate. The configurator can simply replace features when necessary by replacing the corresponding object in the larger object.

When the configurator receives a request to change the features supported by an account object, it will need to check the validity of the request. This means checking the feature inclusion and feature applicability constraints which we implement as logic rules. Each rule is implemented as a rule for the predicate `?featureSet disallowedFor: ?account`.

For the feature inclusion constraints there is a general existing rule which checks facts that correspond to the feature diagram as shown in figure 4.

Figure 5 contains rules for the feature applicability constraints. The first two rules are the implementation of the two constraints we listed in section 2. The other rules are auxiliary rules used in that implementation.

The benefit provided by linguistic symbiosis for writing these rules are that:

- The rules reason about the actual account objects and their subobjects.
- We can easily choose which language or paradigm we want to implement a predicate in.

As to the second benefit, consider these two conditions used in two of the rules in the figure:

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

?featureSet disallowedFor: ?account if
  ?featureSet containsFeature: eur.



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

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

?history badHistory if
  ....
```

Figure 5: Rules for the feature applicability constraints

1. ?account disallowedCreditRating
2. ?featureSet containsFeature: eur

We have chosen to implement the predicate `disallowedCreditRating` as a rule. Thus when the logic inference engine needs to evaluate `?account disallowedCreditRating` it will go on trying to construct a proof by using those rules.

Less obvious from the figure is that `containsFeature:` is implemented as a method. We have chosen to use a method rather than a rule because it is easier to express containment of a feature in the set as a method on the class `FeatureSet` rather than a logic rule. When the logic inference engine needs to evaluate `?featureSet containsFeature: eur` it will thus not find a rule for that predicate and will simply send the message `containsFeature: eur` to the object that is held in the variable `?featureSet`. The method will return true or false, indicating the success or failure of the predicate.

When an interaction is needed between the logic rules and methods that return something other than a boolean, the equality construct can be used. The construct is another addition to the logic language to support the symbiosis. It is used in the figure for example for stating that `?owner` should unify with the object that is returned when sending the message `owner:` to the object held in `account:`, as is done in the first rule for `disallowedCreditRating`.

Having explained the way symbiosis is used to implement the feature inclusion and applicability constraints, we now turn to the next section where we pose some questions on the further applicability of symbiotic languages for component-oriented programming and offer topics for discussion at the workshop.

5 Discussion

Our main question is what the wider applicability of linguistically symbiotic languages could be to Component-Oriented Programming. Because we have presented this idea in the context of GP components, where components are usually understood more as generators or even "anything reusable" [3] than the stricter definition given by Szyperski [9], the question rises whether it has any applications to those kinds of components as well. Certainly the feature implementation parts of the account example are components in this stricter sense, though small-scale. Thus how does this translate to larger scale composition correctness checking of components?

One other area where linguistically symbiotic languages can aid in the reconfiguration of component compositions is in dynamic adaptability of services. Jarir et al. [6] previously presented an adaptable EJB architecture where policies influence the services present in an application. These policies work by monitoring certain system properties and changing policy-bindings when certain conditions on these properties are true. The rules for the policies were written in a sort of XML-based rule language. A full logic language that can interact with the system for retrieving or even reasoning about the properties could make the implementation of the rules simpler.

6 Related work

The work presented here is a further improvement on and proposed application of earlier work we did on linguistic symbiosis [2]. The improvements are mostly in the changing of the syntax of the logic language to get a more transparent interaction with the OO language. In the earlier effort we tried to get such an interaction by mapping message names to predicate names which resulted in less readable rules.

Schaerli [8] has worked on an inter-language bridging between Java and a scripting language for glueing components, Piccola. The aim is also to use a more suitable language for implementing the component compositions while using another for the components themselves. The effort is similar to our earlier work on linguistic symbiosis in that neither language is changed to achieve the interaction, rather the interaction is achieved by a bridging strategy between the two languages that converts objects and message sends to the representation used in the other language.

As it was not our major point we have only briefly described the way we actually compose and recompose features as prototype-based objects. Kniesel has provided a full discussion of how dynamic component adaptation can be achieved with type-safe delegation [7]. Note that his work focused on unanticipated adaptations, while we have focused here on anticipated adaptations or configurations.

7 Summary

We have proposed here to apply the concept of linguistic symbiosis to languages used for composing, reconfiguring or checking the correctness of component compositions. We presented an application in the context of GP generator

components for enforcing the selection of correct features when dynamically reconfiguring the features supported by objects. The constraints that govern the correctness are encoded as logic rules and use the symbiosis mechanism to reason about the state of the objects. We believe this can have wider applications in the Component-Oriented Programming area, such as for monitoring the service-level provided by components and reconfiguring them as needed to meet the required level or checking composition correctness on a larger scale.

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.

[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.

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

[4] Kris Gybels. A survey of Object-Oriented and Logic Multi-Paradigm Programming languages. Technical Report (In preparation), Vrije Universiteit Brussel, 2003.

[5] 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.

[6] Zahi Jarir, Pierre-Charles David, and Thomas Ledoux. Dynamic adaptability of services in enterprise javabeans architecture. In *Proceedings of the Workshop on Component-Oriented Programming*, 2002.

[7] Günter Kniesel. Type-safe delegation for dynamic component adaptation. In *Proceedings of the Workshop on Component-Oriented Programming*, 1998.

[8] Nathanael Schärli. Supporting pure composition by inter-language bridging on the meta-level. Master's thesis, Philosophisch-naturwissenschaftlichen Fakultät der Universität Bern, September 2001.

[9] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1997.

[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.