These are the slides of the talk I gave at the European Smalltalk Users Group Conference 2005, which was based on our paper “Inter-Language Reflection”. I included the notes I wrote for the talk, the actual talk was only slightly different. Unfortunately, I did not write any notes for slides 12 to 18, but you can look at the paper to get the explanation.

© 2005, Kris Gybels. All rights reserved.
Let me begin this talk by giving you this equation as a very brief executive summary of what I’ll be discussing. This equation reflects our conceptual model for understanding inter-language reflection as a combination of linguistic symbiosis and traditional reflection. Traditional reflection is about letting a program manipulate itself as data; inter-language reflection is an extension of this to letting two programs manipulate each other as data. It is *inter-language* reflection because the two programs don’t have to be written in the same language, which is where linguistic symbiosis comes in, which is about letting two programs inter-operate as transparently as possible.

I should also begin by pointing out that what is novel about this work is not the use of inter-language reflection in itself, nor the use of linguistic symbiosis. In fact, in this presentation I’ll be using two examples, each of two languages, between which we’ve previously at the PROG lab for quite some time already introduced a linguistic symbiosis. However, while the relationship of linguistic symbiosis to reflection has already been discussed before, this is the first time its relation to inter-language reflection is discussed. Another contribution of this work is a conceptual model for understanding inter-language reflection, which is also a better model for understanding linguistic symbiosis than the one we’ve been using before.
Our interest in inter-language reflection stems from the fact that for a number of years, PROG and other labs have been doing declarative meta programming. Meta programming in general is about writing programs that are about programs, as opposed to the usual examples of programs about bank accounts etc. While sometimes introduced as an esoteric concept, it’s widely used as in fact most programming tools are meta programs.
In meta programming, the programming language of the meta program tool does not have to be the same as the language of the base program that is manipulated by the tool. Actually, it’s been found that certain programming language paradigms are more suited for certain types of meta programs than other paradigms. One example is TyRuBa, which is similar to Prolog so it is based on the paradigm of logic programming, and is used as a meta language for base programs written in Java. Being based on the logic paradigm makes TyRuBa especially useful for extracting information from the base program.
An important aspect of the way TyRuBa works that we need to look at here is how the base program is represented as data of the meta program. Because TyRuBa is a logic programming language, a Java program needs to be represented as logic facts. In TyRuBa, there is a parser which generates these facts. For the example shown here, a number of logic facts with as name ‘class’ would be generated, and facts with as name ‘method’. These facts together form the meta representation of the base program. In the TyRuBa meta program, there could then be rules which specify when a class is a Visitor class according to the Visitor design pattern. The important aspect here is that, because the facts are generated, there is no connection between the meta representation and the base program it represents. If a class is added to the Java program, this is not automatically reflected in the meta representation, we need to regenerate the facts. Vice-versa, if the TyRuBa program asserts a new ‘class’ fact, the Java program needs to be regenerated from the facts to actually get a new class in it. In reflection terminology, this is dubbed a lack of causal connection between the meta representation and the base program.
Reflection

Reflection is a particular case of meta-programming, which we of course also practice in Smalltalk, where the base program and meta program are actually the same. The meta representation the program accesses is thus a representation of itself. Reflection usually implies a causal connection: when the program changes, the meta representation is also updated, and vice-versa, when the meta representation is changed, the program is changed as well. This is the case in Smalltalk, as illustrated with this example: we can reflectively add a new method to MyOwnClass, and this change is reflected in the selectors list of the class. Now, reflection however also implies that we use the same programming language both for base level programming and meta level programming. This is in contrast with systems like TyRuBa, where we can use a language that is more specialized to meta programming.

```smalltalk
MyOwnClass class>>compile:
...
MyOwnClass class>>selectors
...

MyOwnClass>>makeNewMethod: unarySelector
    MyOwnClass compile: unarySelector asString
^ MyOwnClass selectors size
```
The goal of inter-language reflection then, is to combine the advantages of reflection with those of multi-paradigm meta programming, so that we can use the language of choice for meta programming activities while maintaining a causal connection between base program and its meta representation. Actually, inter-language reflection goes a little further than that, in that we can use either of the two languages to reflect on the other. Thus, in a system like TyRuBa we can use TyRuBa for reflecting on a Java program for meta programming activities where a logic paradigm is more suited, like extracting higher-level information, while on the other hand, Java can be used for meta programming activities where an imperative object-oriented language is more suited, like graphical debugging.
Inter-Language reflection is actually what happens in the combination of SOUL and Smalltalk. SOUL is quite similar to TyRuBa in that it’s also a logic language for reasoning about object-oriented base programs. The important difference is how SOUL gets a meta representation of the Smalltalk program: instead of generating facts, SOUL relies on Smalltalk’s own reflection. This is in turn possible because SOUL has a linguistic symbiosis with Smalltalk. In SOUL, there are no class facts, instead, there is a class logic predicate. There are two implementations for this rule, one to check if a given object is a class, the other to get all the objects representing classes. What it’s important here is that these objects are the reflective meta representations from Smalltalk itself: the linguistic symbiosis between SOUL and Smalltalk means Smalltalk objects can also be used as data values in SOUL. Because Smalltalk reflection is used from SOUL, this inter-language reflection is also automatically causally connected.
As I’ve said in the beginning, the concept of linguistic symbiosis between SOUL and Smalltalk is not new, it originated in the context of RBCL, and was later also explored in combinations of Agora with other languages such as Java. In previous work, it was however used to achieve reflection in RBCL and Agora itself. In all of this previous work, the model that was used to explain the symbiosis was based on one language being implemented in the other. Previously, the same was done to explain the linguistic symbiosis and inter-language reflection relationships in SOUL and Smalltalk as well. We’ve come to realize however that this model confuses these relationships. The contribution of our recent work is that we’ve untangled these relationships by introducing a model where instead of one language being implemented in the other, the two languages are implemented in a common third language. Inter-Language reflection then, becomes a combination of traditional reflection, so that each program can get a self representation, which using linguistic symbiosis can also be passed to the program in the other language. This model also allows us to better understand linguistic symbiosis as a combination of data mapping at the language level, which comes down to a protocol mapping at the language implementation level.
[Didn’t mention Agora is a prototype-based !!!] Data mapping on the language level means it should be possible for programs in the two languages to exchange data, and make it appear as if this is native data. To uphold this appearance, the operations of each language should also be applicable on data from the other language. Here’s an example in Agora which is in symbiosis with Java. The first two lines in this example define two variables frame and ok, in which instances of the Java classes Frame and Button are put. The third line shows that the operation of message sending in Agora also works on Java objects: the message addComponent ‘colon’ is sent to frame. The same thing also happens in the first two lines: the Java classes are made accesible in Agora by sending the message JAVA in all-caps to a string with the name of the class, which returns the Java class itself. The bottom part of the example shows that the reverse is also possible: an Agora object can be passed to Java and be sent messages from Java. The okListener is an Agora object with a method actionPerformedActionEvent ‘colon’. In the last line of this example, it is passed to Java in the addActionListenerActionListener message. In Java, the Agora object appears as an instance of the Java class as declared in its implements method. Its actionPerformed method will then of course be invoked from Java when an action is performed on the Java button.
An example of SOUL of just linguistic symbiosis is that of business rule applications where we encode the business knowledge in logic rules, while domain objects such as customers are written in Smalltalk. This shows some of the mappings that have to occur between SOUL and Smalltalk. For example, when the price of the product needs to be inferred, a logic rule needs to be triggered to get the discount for the product, and somehow the result needs to be returned to Smalltalk. Whether or not a discount is given depends on the customer being loyal to the store, which is the case if he or she has a charge card.
**Data Mapping in SOUL**

```
Product>>priceFor: aCustomer

| discounts |

discounts :=
(SOULEvaluator
  evaluate: 'discount(?customer, ?product, ?discount)
  withArguments:
    (Array with: #customer -> aCustomer
         with: #product -> self)
  valuesForVariable: #discount.

^ price * (100 - discounts max) / 100
```
Protocol Mappings

Language A

Language B

Common Implementation Language

right

left

right

left
send("messageType1Name:...TypenName:",
{right[arg1], ... right[argn]})

\[=\]
result

left[right[x]] = x
lo1 unify: lo2 ⇔ right[lo1] send: #= withArguments: { right[lo2] }
Implementation Mapping

Language A

Language B
receiver.name(argument1, ... argumentn) ==
right[result]

left[receiver].send("nameType1Name:...typenName",
{left[argument1], ...left[argumentn]})
==
result
Conclusion

Multi-Paradigm Meta Programming

Reflection

Inter-Language Reflection

Different Languages

Causal Connection

= Reflection

+ Linguistic Symbiosis

Data Mapping

Protocol Mapping