Symbiotic Reflection between an Object-Oriented and a Logic Programming Language

Roel Wuyts, Stéphane Ducasse
{roel.wuyts | ducasse}@iam.unibe.ch

Software Composition Group
Institut für Informatik
Universität Bern, Switzerland

Abstract. *Meta-programming* is the act of using one system or language to reason about another one. *Reflection* describes systems that have access to and change a causally connected representation of themselves, hence leading to *self-extensible* systems. Up to now, most of the reflective languages have been implemented in the same paradigm. In this paper, we propose *symbiotic reflection* as a way to integrate a meta-programming language with the object-oriented language it reasons about and is implemented in. New to this approach is that any element of the implementation language can be reasoned about and acted upon (not only the self representation), and that both languages are of different paradigms. Moreover, every language implementer that is faced with the problem of allowing the base language to access the underlying meta-language has to solve the problem of enabling entity transfer between both worlds. We propose a uniform schema, called *upping/downing*, to this problem that avoid explicit wrapping or typechecking. We illustrate this with SOUL (the Smalltalk Open Unification Language), a logic programming language in symbiotic reflection with the object-oriented programming language Smalltalk. We show how SOUL does logic reasoning directly on Smalltalk objects, and how to use this to implement *type snooping*.
The contributions of this paper are: (1) the definition of symbiotic reflection, (2) a schema for enabling entities transfer between multiple paradigms, (3) examples of symbiotic reflection.

1 Introduction

In todays rapidly evolving world, development environments need to provide sophisticated tools to inspect, navigate and manipulate software systems. Moreover, developers want design tools that are integrated in their development environment, and expect functionality to keep the design documentation and the implementation consistent. Therefore we integrate a logic programming language called SOUL, in the Smalltalk development environment, and use it as a meta-programming language capable of:

- aiding in program understanding: as logic queries are used to interrogate and match abstract syntax trees (AST) of the software system [22];

– help with forward and reverse engineering: we use the logic programming language to express and extract design information (software architectures, design patterns, UML class diagrams and programming conventions) [9, 23].

Using a declarative programming language to reason about other programs is not new. The well known Lint and its derivatives, for example, use regular expressions as the reasoning engine over source code [6], abstract syntax trees [17] or derived source code information [14, 13, 15]. Other approaches use logic programming languages to do the reasoning [10, 3, 11, 12]. However, new in our approach is that the logic programming language is *fully integrated* with the language we are reasoning about. This integration is based on a new approach to reflective systems, we call *symbiotic reflection. Symbiotic reflection* not only allows one to do pure logic reasoning, but also to:

1. inspect any kind of objects from its implementation language (Smalltalk);
2. write terms that reason about other terms;
3. alter elements of the implementation language.

Hence *symbiotic reflection* differs from 'regular' reflection because it is used in the context of integrating a meta-programming language with the language it is reasoning over, and because these two languages can be of different paradigms. This contrasts with other reflective approaches, that typically use the same languages (for example, Lisp [18], CLOS [7, 1], Smalltalk [5, 16]).

1.1 Introductory Example: Scaffolding Support

In this section we give a concrete example to show the advantages of symbiotic reflection between a logic and an object-oriented programming language. Therefore we use SOUL (Smalltalk Open Unification Language), a logic programming language that is implemented and integrated with the object-oriented programming language Smalltalk. The example shows how to investigate all messages send to a certain variable, and then how to generate methods for all these messages on another class. Hence it implements support for a prototype development approach (as described by scaffolding patterns) where one starts by implementing a first class, and can then use this implementation to generate the skeleton implementation of the class cooperating with this class.

Sends. First of all we write a simple logic rule *sends* that relates three arguments: *?c*, *?rec* and *?sends*. It enumerates in a logic list *?sends* all the messages sent to some receiver *?rec* in the context of a class *?c*. It uses other rules *class* and *method* to state that the variable *?c* should be a class and that *?m* should be a method of that class. Then it uses the *sendsTo* rule (not shown in the implementation here, as this is only a quick example) to enumerate all the sends to the receiver *?rec* in *?sends*[1]:

[1] Some notes on SOUL syntax:

1. the keywords **Rule** , **Fact** and **Query** denotes logical rules facts and queries
2. variables start with a question mark

```
Rule sends(?c, ?rec, ?sends) if
        class(?c),
        method(?c, ?m),
        sendsTo(?m, ?rec, ?sends).
```

We then use this rule to query the Smalltalk system. For example, we can use this rule to find all the messages sent to a variable x in the Smalltalk class *Point*:

```
Query sends([Point], variable(x), ?s
```

However, besides this use of the *sends* rule that gives a list of all the messages sent to x, we can also use it to find in the class *SOULVariable* (the Smalltalk class implementing variables in SOUL) all the expressions (variables, message composition, returns. . .) that invoke the methods *unifyWith:*, and *interprete:*:

```
Query sends([SOULVariable], ?r, <unifyWith:, interprete:>)
```

GenerateEmptyMethod. The second rule is called *generateEmptyMethod*, and generates a Smalltalk method in class *?c* with a given name *?name* (and with an empty implementation). The rule uses an auxiliary predicate *methodSource* that relates the name of a method and a string describing a method with that name (and default arguments, if necessary), that has an empty method body. Then we use a *symbiosis term* represented by [ ] to compile the method *?source* into the class *?c*. The result of the *symbiosis term* is true or false, depending whether the compilation succeeds or not:

```
Rule generateEmptyMethod(?c, ?name) if
        emptyMethodSource(?name, ?source),
        [(?class compile: ?source)  = nil]
```

The following query creates the method *abs* to the class *TestNumber*:

```
Query generateEmptyMethod([TestNumber], abs)
```

Generating the interface.

We can then combine our two rules to generate methods for the Smalltalk class *TestNumber* for all the methods that are send to the variable x in class *Point*:

```
Query sends([Point], variable(x), ?xSends),
        forall(    member(?xSend, ?Sends),
                   generateEmptyMethod([TestNumber], ?xSend))
```

3. terms between square brackets contain Smalltalk code, which can be constants, such as strings or symbols, but also complete Smalltalk expressions that reference logic variables from the outer scope.
4. <> is the list notation

1.2 Example Analysis

This example first of all shows the benefits of using a logic programming language as a meta-programming language to reason about a base language:

– logic programming languages have implicit pattern matching capabilities that make them useful when walking an AST to find certain nodes;
– multi-way: clauses in logic programming languages describe relations between their arguments. These relations can be used in different ways, depending on the arguments passed.
– powerful: it is Turing computable. We used it to express and extract design information such as design patterns or UML class diagrams from the source code [22, 23].

More importantly, it also demonstrates the different kinds of reasoning and reflection available:

1. *Introspection.* SOUL terms can reason about other SOUL terms (as is shown in the query where we use *SOULVariable*).
2. not shown in this example, but later on in the paper, is the implementation of second-order logic predicates like *findall, forall, one, calls,* ... in SOUL itself. This shows how logic predicates can change the data of the SOUL interpreter from within SOUL itself;
3. *Symbiotic Introspection*: we also do logic reasoning directly over Smalltalk objects, i.e., on the meta-language itself. In the example we use Smalltalk classes, that are then inspected to get the methods they implement. It is important to note here that these are the Smalltalk objects themselves that are used, and not decoupled representations;
4. *Symbiotic Intercession*: we use the logic programming language to modify code in the implementation language. Thus, not only can we inspect Smalltalk objects, we can also change them. For example, the *generateEmptyMethod* rule adds methods to a class. Because the class that is passed is the actual Smalltalk class, adding this method immediately updates the base language.

In symbiotic reflection, as the meta-language implements the base language and the base language can reason about and act on the meta-language, both the base language and the meta-language can then act and reason on each other.

In the rest of this paper we describe how to obtain symbiotic reflection between two languages from different paradigms, and how it is implemented in our logic programming language SOUL. We end the paper with some examples: a type snooper and the definition of some second-order logic predicates.

2 Reflective Interpreters

In this section we give an overview of *non-reflective interpreters*, classic *reflective interpreters* and *symbiotic reflective interpreters*, and their differences. In the

following sections we then discuss the implementation of a symbiotic interpreter in general, and the particular case of our example language, SOUL.

First of all we want to establish some classic terminology. When implementing an interpreter, the language implementing the interpreter is the *meta-programming language* (hereafter called M), and the interpreted language is the *base language* (hereafter called B). The meta-programming language interprets the program that implements the base language. Both the meta-programming language and the base language manipulate certain data. The difference between a non-reflective, a reflective and a symbiotic reflective interpreter lies in the data they manipulate.

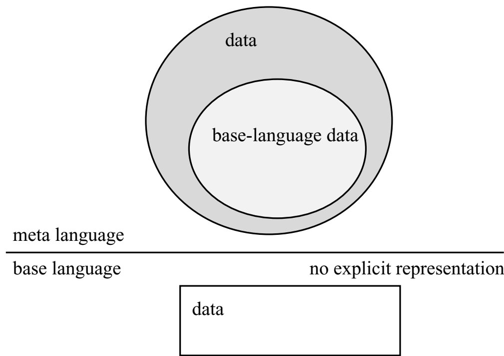

Fig. 1. *A non-reflective interpreter. The base language can only manipulate base level information and not meta-level information.*

Non reflective interpreter. A *non-reflective* interpreter is a program written in the meta-programming language, that uses its own data and does not interact with its meta-programming languageas shown in Figure 1. Thus, interpreting an expression in a non-reflective interpreter only requires to manipulate base language entities at the meta-level. As the interpreter is built in the meta-language, we have $arg1, \ldots, argn \in Bininterpret(arg1, arg2 \ldots argn)$.

Reflective interpreter. Before we look at a reflective interpreter, we define what is meant by *causally connected*, and by a *reflective system*:

Definition: causally connected *A computational system is* causally connected *to its domain if the computational system is linked with its domain in such way that, if one of the two changes, this leads to an effect on the other [8].*

Definition: reflective system *A* reflective system *is a causally connected meta system that has as base system itself [8].*

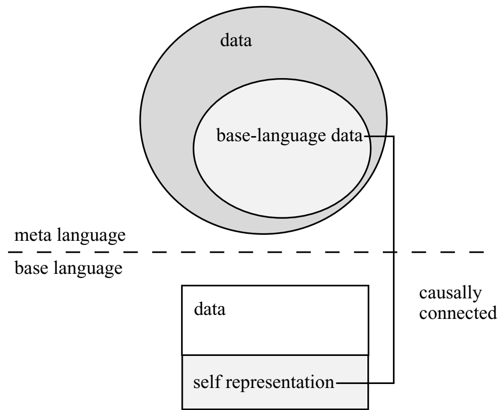

Fig. 2. *A Reflective Interpreter. The base language can access and act on its self-representation*

Definition: Reflection. *Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation:* introspection *and* intercession. *Introspection is the ability for a program to observe and therefore reason about its own state. Intercession is the ability for a program to modify its own execution state or alter its own interpretation or meaning. Both aspects require a mechanism for encoding execution state as data; providing such an encoding is called* reification. *[1]*

As shown in Figure 2, a reflective interpreter can access and manipulate two kinds of data: (1) the base level data and (2) a causally connected representation of itself, called the *self representation* [19].

During the interpretation the arguments can be from both levels (but the meta-entities have to be part of the data implementing the base-language). So when interpreting an expression:

$interpret(arg_1, arg_2 \dots arg_n)$

the arguments $arg_1, \dots, arg_n$ are

- base language entities treated at the meta-level,
- self-representing meta-entities.

Symbiotic reflective interpreter. A symbiotic reflective interpreter as shown in figure 3 is a reflective interpreter that, in addition to being able to manipulate its self-representation can also manipulate the meta-language. As the meta-language implements the base language and the base language can reason about and act on the meta-language, both base language and meta-language can then act and reason on each other.

For example, in the SOUL expression:

method([Array], ?m)

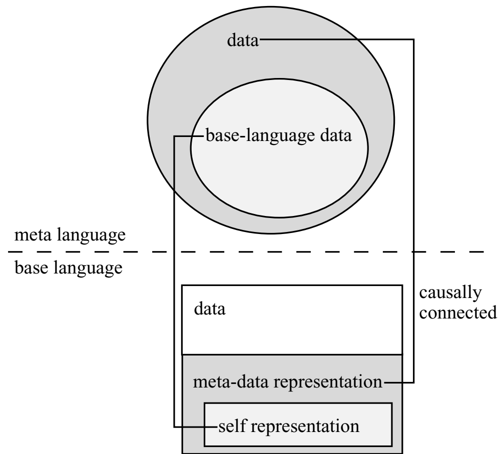

Fig. 3. *A Symbiotic Reflective Interpreter. From the base language it is now possible to access and manipulate the base language self representation and also the meta-level representation*

the interpreter manipulates *Array* (a Smalltalk entity that has nothing to do with SOUL's implementation).

For example, in the SOUL expression:

method([SOULVariable], ?m)

the interpreter manipulates *?m*, a variable term (a base language entity) and *SOULVariable* (a meta-entity from SOUL's Smalltalk implementation, part of the self-representation).

Different meta and base languages. We stress that reflective systems that are written in the same language are in symbiotic reflection because of their uniformity. However, distinguishing symbiotic reflection from reflection is mandatory when different languages are involved where the meta-language can be modified from the base language. The next section shows how to solve the problems that arise during the interpretation of the manipulated entities.

3 Symbiotic Reflection between Two Languages

In this section we start presenting the problems that occur when the base language has to be able to manipulate its meta-language. Then we show how the upping/downing schema proposes a uniform solution.

3.1 Problems with Handling Objects from Two Different Worlds

Enabling the reflection between two languages requires that entities of both languages can be manipulated in each language. When the two languages are

the same, this is not a problem because all the entities share a common data structure or, in the case of an object-oriented reflective language, a polymorphic representation. For example, in Smalltalk, *instVarAt:* reflective method allows one to access the instance variable of any object because it is defined on the class *Object*.

In our case the logic programming language is implemented in the object-oriented programming language, and represents and acts on the object-oriented one. The logic engine is able to manipulate objects as terms and the terms are manipulated as objects. Suppose SOUL would not use the upping/downing schema we present further on, then lots of (implicit or explicit) type checks would be needed to check every time whether we are using a logic term or an object.

A concrete example. In the logic programming language we might have a *unify* predicate to unify two arguments. This predicate can be called in different ways, both with objects as with terms:

Query unify(?c, foo(bar)).

Query unify(?c, [Array]).

This predicate has to be implemented somewhere in the object-oriented programming language. So, there is some method that implements this logic unification of two arguments. However, as we see in the logic code, the arguments can be instances of the classes implementing logic terms (like *?c* or *foo(bar)*) as well as objects (like *Array*), that have nothing to do with the implementation of the logic interpreter.

The problem is that the interfaces of these classes differ. The classes implementing logic interpretation will typically know how to be unified and interpreted logically, whereas regular objects do not. Possible solutions are:

– All methods in the logic interpretation that come in contact with logic terms need to do an explicit typechecking and conversion in the case of a dynamically typed object-oriented programming language or provide several methods with different types in the case of a statically typed object-oriented programming language, or
– implement everything on the root class, so that objects can be used as terms and vice versa.

Neither solutions are satisfactory. In the first one lots of different type-checks have to be done throughout the implementation of the logic interpreter. For the second solution we effectively have to change the implementation language and implement the complete behaviour for the logic interpretation on the root class.

We would like to stress that such a *transfer* of entities between languages has to be addressed in any language where data structures from the meta-programming language can be manipulated from the base language. At the worse the programmer has to be aware that he is manipulating implementation entities and has to interpret or wrap them himself.

3.2 The Upping/Downing Schema

A unified and integrated solution is possible. In our case, it enables objects to be manipulated as logic terms and terms as objects. To explain such a schema we have to introduce two levels: the up level and the down level.

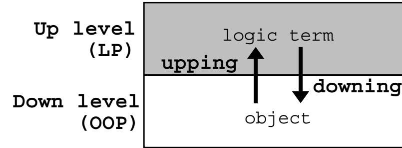

Fig. 4. *The up-down schema allows the uniform manipulation of entities. In our context, it lets Smalltalk objects be directly accessed in SOUL.*

Symbiotic reflection implies that *both* languages play the base and the meta-language role. The role depends on the view we have on the overall system. From a user point of view, the logic programming language representing and manipulating the object-oriented language acts as a meta language while the object language acts as a base language. From the interpreter point of view, as the logic programming language is implemented in the object-oriented one, the object-oriented one is the meta-language and the logic programming the base. Hence, it is not clear what we mean by 'meta level' or 'base level' in this context, so from now on we consider *two* conceptual levels as shown in figure 4.

1. the *down* level is the level of the implementation language of the logic programming language (the object-oriented programming language);
2. the *up* level is the logic programming language level being evaluated by the *down* (object-oriented programming language) level.

Enabling the access and manipulation of down level structure (the object-oriented programming language) from the up level (the logic programming language) in a unified way is possible by following the simple transfer rule: upping a down entity should return an upped entity and downing an upped entity should return a down entity. Applied to SOUL, this rule reads: upping an object should return a term and downing a term should return an object.

This is expressed by the following rules where T represents the set of terms and O the sets of objects, $wrappedAsTerm$ is a function that wraps its argument into a term and $implementationOf$ is a function that returns the data representing its argument.

$up : O \rightarrow T$

- (1) $x \in T, up(down(x)) = x$
  For example in SOUL, $up(implementation(?c)) = ?c$
- (2) $x \notin T, up(x) = wrappedAsTerm(x)$
  For example in SOUL, $up(1) = [1] = wrappedAsTerm(1)$, where [1] is the logic representation of a term wrapping the integer 1.

$down : O \rightarrow T$

- (3) $x \in T, down(x) = implementationOf(x)$
  For example in SOUL, $down(?c) = aVariableTerm$, the smalltalk object of the logic variable ?c.
- (4) $x \notin T, down(up(x)) = x$
  For example in SOUL, $down([1]) = 1$, where [1] is the logic representation of a term wrapping the integer 1.

The transfer rules (1) and (4) are limiting the meta-level to one level. The transfer rule (2) expresses that upping a plain object results in a wrapper that encapsulates the object and acts a term (and so can be logically unified and interpreted). The transfer rule (3) expresses that downing an ex-nihilo logic term to return the object implementing that term.

The upping/downing schema presented above is analogous to that described in the PhD dissertation of Steyaert as the core implementation mechanism for a framework for open designed object-oriented programming languages [21]. The implementation of the object-based object-oriented programming language *Agora* uses the *up/down* mechanism to get reflection with its object-oriented implementation language (Smalltalk, C++ or Java) [4]. However, in the context of this paper we use it as the cornerstone to get reflection between two languages from *different paradigms*.

3.3 Using the Upping/Downing Schema

We now use the upping/downing schema to implement the interpretation in a straightforward way without having to typechecking entities.

When we evaluate a logic expression to unify terms, we are clearly reasoning at the logic level (the up level). Hence we conceptually think in terms of terms and interpretation, and expect the result to be a logic result (a logic failure or success, with an updated logic environment containing updated logic bindings). However, the interpreter is a program in the object-oriented programming language (the down level), so somehow this has to be mapped, taking care that everything is interpreted at the *down level*.

Generally speaking, to interpret an up-level expression:

- we down all elements taking part in that expression;
- we interpret the expression at the down level, and obtain a certain down-level result;
- we up this result.

This can be expressed the following way:

Given t a logic term and θ a logic environment,

$$< t > , \theta = \{v \rightarrow w, \ldots\} = up\big(down(t).\text{interpreteIn}(down(\theta))\big)$$

Example 1. Let us look at the interpretation of the following SOUL expression (See section 1.1):

sends([SOULVariable], variable(name), ?xSends)

This expression consists of a compound term, with three arguments. The Smalltalk object representing this logic expression is a parse tree that consists of an instance of class SOULCompoundTerm, that holds on to its arguments.

Interpreting the logic expression in a logic context comes down to sending *interprete:* to the parse tree at the Smalltalk level (taking the logic context as an argument). Therefore we down the parse tree and the logic context before sending it *interprete:*. The result of sending *interprete:* is a Smalltalk object, and an updated logic context (containing bindings for the variable *?xSends*). This is then upped to get the logic result.

Example 2. Because of the explicit upping and downing, the evaluation works as well for objects as for terms, contrary to non-reflective systems. Let's evaluate the following expression:

[(?class compile: ?source) = nil]

in a logic environment θ where variable *?class* is bound to *[TestNumber]*, and variable *?source* is bound to *'abs "empty method source"'*. This is depicted in figure 5.

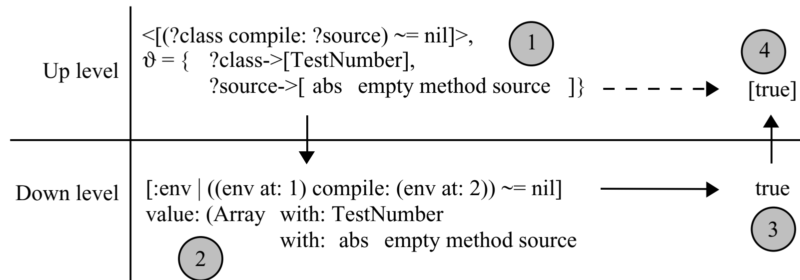

Fig. 5. *Interpreting a* symbiosis term *in a logic environment* θ

To interpret this expression (step 1 in the figure) we send *interprete:* to the downed parse tree representing this logic expression, with as argument the downed environment(step 2 in the figure). This results in the following Smalltalk expression being evaluated:

TestNumber compile: 'abs "empty method source"') = nil

This piece of Smalltalk code compiles a method in the class *TestNumber*. The source describes a method called *abs*, that contains no statements, but just some comment. The result of sending *compile:* is nil if something went wrong, or the compiled method if everything went ok. So, the final result of the complete expression is the Smalltalk object *true* if the method was successfully compiled, and false otherwise (step 3 in the figure). This result is upped to get a result in SOUL: a success or a failure (step 4).

3.4 The Symbiosis Term

Symbiotic reflection requires one base symbiotic operator that makes the bridge between the base level and the meta-level. The SOUL language construct enabling symbiosis is the *symbiosis term* that allows one to use Smalltalk code (parametrized by logic variables) during logic interpretation. The *symbiosis term* is a logic term that wraps Smalltalk objects and message sends in the logic programming language[2]. From the users point of view the *symbiosis term* takes the form of writing a regular Smalltalk expression that can contain logic variables as receivers of messages, enclosed within square brackets as shown by the examples in section 1.1.

4 Symbiotic Reflection Examples

Throughout this paper we described concrete examples written in our symbiotic reflective language SOUL (Smalltalk Open Unification Language). SOUL is a logic programming language (analogous to Prolog [2, 20]) that is implemented in, and lives in symbiosis with, the object-oriented programming language *Smalltalk*. Using SOUL tools were built that use logic reasoning directly in the development environment, while ensuring that they always work on the current version of the source code [23].

In this section we give examples of the symbiotic reflection. We first look at the implementation of a *type snooper*, and then we show some implementations of second-order predicate.

4.1 The Type-Snooper

In SOUL we implemented a lightweight type-inferencer for instance variables, that uses the messages send to an instance variable in the context of a class to determine an interface that possible types must comply to. Then we find all the classes that understand all these messages to deduce the possible types. This basic scheme was extended taking programming conventions into account [23].

Using symbiotic reflection we now show how to integrate *type snooping* with this lightweight type-inference. *Type snooping* uses the fact that in the Smalltalk development environment objects exist from the class we want to find types of instance variables for. Hence, by looking at these instance variables we find collections of existing types. The following rules use symbiotic reflection to interrogate our Smalltalk development environment for such instances, and to get their types. Then we extract the types for the instance variables we are interested in. This is yet another set of possible types, that we can integrate with the rest of our typing rules.

[2] For Prolog users: despite its name, a *symbiosis term* can be used both as term and as predication.

```
Rule objectsForVar(?class, ?var, ?objects) if
        class(?class),
        instVar(?class, ?var),
        instVarIndex(?class, ?var, ?index),
        generate(   ?objects,
                    [(?class allInstances collect: [:c |
                          c instVarAt: ?index]) asStream]).

Rule snoopTypeInstVar(?class, ?var, ?types) if
        findall(    ?cl,
                    and(  objectsForVar(?class, ?var, ?o),
                          objectClass(?cl, ?o)),
                    ?allTypes),
        noDups(?allTypes, ?types).
```

Without symbiotic reflection integrating such support is not possible, because we can not reason about the elements of our base language. In symbiotic reflection we have the objects, and can use them as such, so that we can directly reuse them in the logic interpretation.

4.2 Second-order logic

This example shows how to write second-order logic predicates using the *symbiosis term*. Therefore we reify two concepts that are important during the evaluation of a logic term: the logic repository and the logic environment that holds on to the bindings. We chose to make these two concepts available in the *symbiosis term*, under the form of two hardcoded variables: *?repository* and *?bindings*. The *?repository* variable references the logic repository used when interpreting the *symbiosis term*. The *?bindings* variable holds the current set of bindings. This simple addition makes it possible for a *symbiosis term* to inspect and influence its interpretation. As an example we give the implementation of three widely used logic predicates: *assert*, *one* and *call*. The *assert* predicate adds a new logic clause to the current repository. The *one* predicate finds only the first solution of the term passed as argument. If this first solution is found, the bindings are updated and the predicate succeeds, otherwise the predicate fails. The *call* predicate is analogous to the *one* predicate, but does not keep the results. Hence it just needs to succeed when the argument term has at least one solution:

```
Rule assert(?clause) if
        [?repository addClause: ?clause ].

Rule one(?term) if
        [       | solution |
                solution := ( ?term resultStream: ?repository ) next.
                solution isNil
                        ifTrue: [false]
                        ifFalse: [ ?bindings addAll: solution. true ]
```

```
        ].

    Rule call(?term) if
        [(?term resultStream: ?repository) next isNil not]
```

Speaking in reflection terminology, the two hardcoded variables *?repository* and *bindings* are a causally connected self-representation. Therefore the *symbiosis term* (and hence SOUL) can reason about and even alter a part of its implementation.

5 Conclusions

In this paper we presented *symbiotic reflection symbiotic reflection* as a way to integrate one language (the up-level language) with another language (the down-level language) it reasons about and is implemented in. The benefit is that the up-level language can not only reason about its self-representation (as is the case with classic reflection), but on the complete down-level language. Symbiotic reflection was illustrated concretely with the object-oriented programming language SOUL, a logic programming language in symbiotic reflection with Smalltalk:

– *Introspection* SOUL terms can do logic reasoning about other SOUL terms;
– *Reflection* SOUL predicates can change data of the SOUL interpreter from within SOUL;
– *Symbiotic Introspection*: SOUL can do logic reasoning about any Smalltalk object;
– *Symbiotic Intercession*: SOUL can do logic reasoning to modify code in the implementation language, and this immediately impacts the implementation language.

To show the benefits of symbiotic reflection we expressed three non-trivial concrete examples in SOUL. We wrote logic predicates implementing second-order logic operations in SOUL, provided prototype development support and integrated lightweight type-inference with type snooping.

6 Acknowledgments

Thanks to everybody who contributed to this paper: the ex-colleagues from the Programming Technology Lab (where Roel Wuyts did his phd of which this paper is a part) and the people of the Software Composition Group. All these people are thanked for their positive feedback while doing the research and writing this paper.

References

1. D.G. Bobrow, R.P. Gabriel, and J.L. White. Clos in context – the shape of the design. In *Object-Oriented Programming : the CLOS perspective*, pages 29–61. MIT Press, 1993.
2. W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, 1981.
3. Roger F. Crew. Astlog: A language for examining abstract syntax trees. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, 1997.
4. Wolfgang De Meuter. Agora: The story of the simplest mop in the world - or - the scheme of object-orientation. In *Prototype-based Programming*. Springer Verlag, 1998.
5. Brian Foote and Ralph E. Johnson. Reflective facilities in smalltalk-80. In *OOPSLA 89 Proceedings*, pages 327–335, 1989.
6. S. C. Johnson. Lint, a C program checker. *Computing Science TR*, 65, December 1977.
7. Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
8. Patty Maes. *Computational Reflection*. PhD thesis, Dept. of Computer Science, AI-Lab, Vrije Universiteit Brussel, Belgium, 1987.
9. K. Mens, R. Wuyts, and T. D'Hondt. Declaratively codifying software architectures using virtual software classifications. In *Proceedings of TOOLS-Europe 99*, pages 33–45, June 1999.
10. Scott Meyers, Carolyn K. Duby, and Steven P. Reiss. Constraining the structure and style of object-oriented programs. Technical Report CS-93-12, Department of Computer Science, Brown University, Box 1910, Providence, RI 02912, April 1993.
11. Naftaly H. Minsky. Law-governed regularities in object systems, part 1: An abstract model. *Theory and Practice of Object Systems*, 2(4):283–301, 1996.
12. Naftaly H. Minsky and Partha Pratim Pal. Law-governed regularities in object systems, part 2: A concrete implementation. *Theory and Practice of Object Systems*, 3(2):87–101, 1997.
13. G. Murphy and D. Notkin. Lightweight source model extraction. In *Proceedings of SIGSOFT'95, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 116–127. ACM Press, 1995.
14. G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of SIGSOFT'95, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28. ACM Press, 1995.
15. G. C. Murphy. *Lightweight Structural Summarization as an Aid to Software Evolution*. PhD thesis, University of Washington, 1996.
16. Fred Rivard. Reflective Facilities in Smalltalk. *Revue Informatik/Informatique, revue des organisations suisses d'informatique. Numéro 1 Février 1996*, February 1996.
17. D. Roberts, J. Brant, R. Johnson, and B. Opdyke. An automated refactoring tool. In *Proceedings of ICAST '96, Chicago, IL*, April 1996.
18. B. Smith. Reflection and semantics in lisp. In *Proceedings of POPL'84*, pages 23–3, 1984.
19. Brian C. Smith. *Reflection and Semantics in a Procedural Language*. PhD thesis, MIT, 1982.
20. L. Sterling and E. Shapiro. *The art of Prolog*. The MIT Press, Cambridge, 1988.

21. Patrick Steyaert. *Open Design of Object-Oriented Languages, A Foundation for Specialisable Reflective Language Frameworks.* PhD thesis, Vrije Universiteit Brussel, 1994.
22. R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings TOOLS USA'98, IEEE Computer Society Press*, pages 112–124, 1998.
23. Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation.* PhD thesis, Vrije Universiteit Brussel, 2001.