Towards Linguistic Symbiosis of an Object-Oriented and a Logic Programming Language

Johan Brichau[1], Kris Gybels[1], and Roel Wuyts[2]

[1] Programming Technology Lab,
Vakgroep Informatica,
Vrije Universiteit Brussel, Belgium
{johan.brichau, kris.gybels}@vub.ac.be
[2] Software Composition Group,
Institut für Informatik,
Universität Bern, Switzerland
wuyts@iam.unibe.ch

Abstract. Reflective systems have a causally connected (metalevel) representation of themselves. Most reflective systems use the same language to reason about their metalevel representation as the language that is used to reason about their domain. In *symbiotic reflection* a different language is used at the metalevel. The practical usability of this symbiotic reflection is enhanced if a *linguistic symbiosis* is accomplished that transparantly integrates both languages. Implementing such a linguistic symbiosis is relatively straightforward if the meta language and the base language share the same programming paradigm. The problem becomes far more complex when the paradigms differ. This paper describes the extension of the symbiotic reflective system SOUL with a linguistic symbiosis between a logic meta language and an object-oriented base language.

1 Introduction

Reflection is a technique that realizes flexible systems. This is because a reflective system can manipulate data that is a representation of its own computation (called causally-connected self-representation). As such, a reflective system can adapt its own computation. For example, truly reflective programming languages, such as Smalltalk or CLOS, allow to introduce new language features or change the internal workings of existing ones.
Typically, the causally-connected self-representation (CCSR) of a programming language is expressed in the same programming language. There also exist reflective languages in which the CCSR is expressed in a different programming language (e.g. Agora [5] and RbCl [2]). But this

[1] Johan Brichau and Kris Gybels are research assistants of the Fund for Scientific Research - Flanders (Belgium) (F.W.O.)

language still adheres to the same programming paradigm as the base language (e.g. object-orientation) which allows for a relatively straightforward integration. This integration is called *linguistic symbiosis* [2], which means that program elements from one language can be used transparently in the other language, and vice-versa. It also means that one language is actually implemented in the other language, which enables mutual reflective capabilities.

In this paper, we describe the linguistic symbiosis between a logic and an object-oriented programming language. The linguistic symbiosis is particularly hard to achieve because these paradigms are fundamentally different. In general, object-oriented programs consist of objects, containing state, that communicate through messages. All control flow is explicitly programmed. Logic programs consist of rules that describe how a certain fact is true under some conditions. The control flow is implicit, i.e. the interpreter will prove that a certain fact is true by proving its conditions. Also, a message send is fundamentally different from a predicate call. A message always returns a single result and must always be provided with a fixed number of arguments while a predicate can return multiple results for multiple variables that were left unbound in the predicate call. Numerous object-oriented extensions to logic programming languages exist. Basically, these extensions enhance logic programming with modularization, inheritance and late binding but the overall paradigm remains logic programming. This is fundamentally different from extending an object-oriented language with a logic programming paradigm.

Wuyts previously presented a system in which symbiotic reflection was introduced between an object-oriented base language and a logic meta language [8]. In the resulting system (SOUL), the CCSR is expressed in a logic language and it has shown to be particularly useful [3, 4, 7, 6] because logic programming languages are better suited to express reasoning algorithms such as type inferencing, design pattern extraction, architectural conformance checking, etc. . . .

The current implementation of SOUL already allows multi-paradigm programming with the logic and object-oriented paradigms because values can be exchanged between programs written in the different paradigms. But SOUL does not introduce a linguistic symbiosis. We cannot invoke programs implemented in the object-oriented language in the same way as programs implemented in the logic language. To solve this problem, we introduce a complete linguistic symbiosis between a logic and an object-oriented programming language, and apply this to SOUL. This allows object-oriented programs to *transparently* use libraries of logic programs (designed for reflective programming) during method execution. The implementer of the library can then make use of the full power of the logic environment while the regular object-oriented programmer is not exposed to it.

This paper is organized as follows. In the next section, we describe the SOUL symbiotic reflective system and how SOUL and Smalltalk are cross-bound. Section 3 describes extensions to SOUL to accomplish a linguistic symbiosis between SOUL and Smalltalk. In section 4 we describe an example application and section 5 discusses the open issues.

2 SOUL

SOUL (Smalltalk Open Unification Language) is a logic programming language implemented in Smalltalk. But SOUL offers more than the ability to write Prolog-like logic programs: it supports the embedding of Smalltalk expressions in a logic program, which are executed as part of the inference process. These Smalltalk expressions can use logic variables and Smalltalk objects can be unified with logic variables. Furthermore, logic programs can be invoked from within a Smalltalk program by invoking a query. This section shows how this integration of Smalltalk and SOUL is accomplished. A more detailed discussion can be found in [7]. We first give a brief overview of how logic programs in SOUL differ from Prolog programs.

2.1 Differences with Prolog

The differences between Prolog and SOUL are mostly syntactic.

Variables in SOUL start with a question mark (e.g. `?var`).

Lists are enclosed in '`<`' and '`>`' (e.g. `<1,2,3,4>`).

Rules are written with the keyword '`if`' instead of a '`:-`' symbol.

Modules are used to encapsulate logic declarations (facts and rules). Each logic declaration belongs to a module and is only visible in the module where it is defined [3].

Querying other modules A rule in one module can invoke a query in another module using the '`.`' operator. For example, invoking a query `'myQuery(?x)'` in a module called `'Mymodule'`, is written as follows: `if Mymodule.myQuery(?x)`

2.2 Cross-binding Smalltalk and SOUL

Smalltalk in SOUL Invocation of Smalltalk programs from within logic SOUL programs is accomplished through special constructs called *Smalltalk term* and *Smalltalk clause*. Furthermore, Smalltalk objects are treated as constants. Since everything in Smalltalk is an object, these special constructs are the only new kind of logic terms that are a direct result of the integration. We now describe each construct in more detail.

Smalltalk term This is a term that contains a Smalltalk expression enclosed in square brackets `[...]`. Each time the inference engine has to unify this term with another term, the expression is evaluated and the resulting Smalltalk object is used to complete the unification (such as binding it to a logic variable). As such, the real Smalltalk objects themselves can be bound to logic variables (how this is done, is explained later). Furthermore, the Smalltalk expression is allowed to contain logic variables. For example, the SOUL declaration `'object([ Object new])'` is a logic fact with a Smalltalk term as its single argument that contains an instance of `Object` (not just a notation for it).

[3] Modules have no syntactic notation. How they are built is out of the scope of this paper.

Smalltalk clause This is a predication that is syntactically the same as a Smalltalk term, except that the embedded expression must evaluate to true or false. For example, the SOUL declaration `'smaller(?x,?y) if [?x < ?y]'` is a logic rule that uses a Smalltalk clause to compare the values of `?x` and `?y`.

Smalltalk object Objects can get bound to logic variables, as a result of the unification of a Smalltalk term with a logic variable. We extend the unification-scheme of SOUL to include Smalltalk objects such that they are treated as constants. This means that objects only unify with (free) variables and themselves.

generate/2 predicate The `generate/2` predicate is a predicate that decomposes a Smalltalk collection object into subsequent results of a variable (similar to what `member/2` does for logic lists). For example the query `'if generate(?x,[Smalltalk allClasses])'` returns many results for `?x`, i.e. all classes in the Smalltalk image. This is because the expression `Smalltalk allClasses` returns a collection object and the generate predicate subsequently binds `?x` to each element of this collection.

SOUL in Smalltalk Since SOUL is implemented in Smalltalk, Smalltalk programs can use the SOUL implementation to start logic queries. Without the transparent invocation mechanism presented later on, a SOUL query has to be invoked by sending a message to the class `SOULEvaluator` and iterate over the returned results after the evaluation. The following is part of a Smalltalk method containing the invocation of a query:

```
...
a := 1 + 2.
results := (SOULEvaluator eval:'if member(?x,<1,2,3,4>)'
                                withArgs: #((x a)) ) allResults.
results succes ifTrue:[...] ifFalse:[...]
...
```

This example shows how a query `member(?x,<1,2,3,4>)` should be invoked where the value of the logic variable `?x` is the value of the Smalltalk variable `a`. When the query succeeds, the evaluator will return an instance of the `SoulResults` class indicating the success of the query and holding a collection of all successful variable bindings. All the values of these bindings get converted from logic terms to Smalltalk equivalents as follows so they can be easily manipulated by the Smalltalk programmer:

Smalltalk objects are trivially mapped onto themselves.
Logic constants are mapped onto a Smalltalk symbol.
Logic integers are mapped onto a Smalltalk integer.
Logic lists are mapped onto Smalltalk `OrderedCollection`s.
Logic functor terms are mapped onto a special class `CompoundTerm`

When the query fails, the evaluator also returns an instance of the `SoulResults` class indicating that the query failed.

2.3 Symbiotic Reflection

The SOUL interpreter is a reflective system because it is implemented in Smalltalk and can thus use the Smalltalk meta-object protocol to

investigate and adapt its own implementation. By allowing the use of Smalltalk objects in the SOUL language, the SOUL programmer also has access to this MOP and can reify every Smalltalk program and thus also SOUL itself. Hence, SOUL is in symbiotic reflection with Smalltalk. For example, the rules that reify Smalltalk classes to logic declarations are:

```
class(?x) if
  not(var(?x)),
  generate(?x,[Smalltalk allClasses]).

class(?x) if
  var(?x),
  [Smalltalk allClasses includes: ?x].
```

The first rule implements one possible usage of the `class/1` predicate where the variable `?x` is not bound to a value. Therefore, the rule will subsequently bind `?x` to a Smalltalk class. The second rule implements the case where `?x` is bound to a value and therefore, it will check if this value is a Smalltalk class.
Hence, we can invoke a query to gather all classes or invoke a query to check if a class `Symbol` exists:

```
if class(?x).
if class([Symbol]).
```

With the Smalltalk system reified in SOUL, the power offered by logic programming was used to express design patterns, programming conventions and software architectures, to name but a few [3, 4, 7, 6].

3 Towards Linguistic Symbiosis

A *linguistic symbiosis* [2] for Smalltalk and SOUL means that a Smalltalk program can transparently call a SOUL program as if it was a Smalltalk program and a SOUL program can transparently call a Smalltalk program as if it was a SOUL program. The result is that a meta programmer can use the full power of symbiotic reflection while a Smalltalk programmer can use the reflective facilities offered by SOUL without knowing that they are actually implemented in logic programs.
In the integration of Smalltalk and SOUL as described above, it is actually explicitly coded where a logic program and where a Smalltalk program is used:

SOUL to Smalltalk Calling a Smalltalk program from SOUL is made explicit through the use of a Smalltalk term or clause (using a Smalltalk term or clause).

Smalltalk to SOUL Calling a SOUL program from within Smalltalk is made explicit by sending a query message to the SOULEvaluator class.

To accomplish linguistic symbiosis between the logic and object-oriented languages, we have to map the main concepts of both paradigms on each other. Basically, we chose to map message sends in the object-oriented paradigm on queries in the logic paradigm. So, the invocation of a query should be the same as a message send and vice-versa. Therefore, we propose the following mapping:

1. Smalltalk classes `<=>` SOUL modules;
2. Smalltalk message sends `<=>` SOUL predicates;
3. Smalltalk collections `<=` SOUL query results.

In the following sections we explain each mapping in more detail.

3.1 Smalltalk classes and SOUL modules

The namespace of all Smalltalk classes and the SOUL namespace of all modules is combined into one namespace which is accessible from both the SOUL and Smalltalk environment. This results in a combined dictionary of (logic) modules and (object-oriented) classes. Smalltalk classes encapsulate methods, while SOUL modules encapsulate logic facts and rules. This difference becomes apparent when sending messages to or invoking queries on both Smalltalk classes and instances as well as SOUL modules.

3.2 Querying Smalltalk objects

In order to be able to transparently send messages from within a SOUL program, message sends should be expressed as queries. As a result, messages need a representation in the form of a predicate. Therefore, we define a straightforward mapping of messages to predicates. The predicate name is the message selector and the predicate arguments are the message arguments with an extra last argument for the return value of the message. For example, the SOUL query:

```
if Array.new:(10,?instance),
  ?instance.at:put:(1,2,?returnvalue)
```

corresponds with the message(s):

```
instance := Array new:10.
returnvalue := instance at: 1 put: 2.
```

In this example, an array of size 10 is created and stored in the variable `instance`. Afterwards, the integer 2 is stored at position 1 and the returning value of this message is stored in the variable `returnvalue`.

3.3 Sending messages to SOUL modules

Conversely, in order to be able to transparently invoke SOUL queries from within Smalltalk methods, a query should be expressed as a message send. As a result, SOUL predicates should have a Smalltalk message representation. Therefore, we define a mapping of logic predicates to Smalltalk messages, using Smalltalk's keyword messages.
Keyword messages consist of a selector where the arguments are interleaved with the keywords of the selector and keywords always end with a colon (e.g: `at: index put: anObject`). Because such keyword messages allow for a non-ambiguous mapping of logic queries to Smalltalk messages, we require that the names of all logic predicates use the signature of a keyword message. This means that the predicate name should consist of as many keywords as the predicate's number of arguments.

Furthermore, this mapping also requires that the keywords are unique for each predicate that is defined in a module. The benefit of using keyword messages is explained later. Some example logic declarations that use this naming convention are:

```
add:with:to:(?x,?y,?result) if ...
method:inClass:(?method,?class) if ...
class:(?class) if ...
```

Of course, the problem is that a logic query can return multiple results for multiple variables, as opposed to a Smalltalk method, which always returns only one single result. Moreover, the same logic predicate can be used in multiple ways (i.e. with all arguments bound, all arguments unbound or only some of them bound). These problems are addressed in the following subsections.

Translating multiway predicates to messages To translate the multi-way property of logic predicates to Smalltalk, a logic module automatically understands a message for each way in which a logic predicate in this module can be used. As such, a single predicate in a logic module (possibly implemented by different logic facts and rules) corresponds to a set of Smalltalk messages that can be sent to the logic module. Because the name of a predicate uses a keyword message signature, the signatures of the corresponding Smalltalk messages can be easily derived from the name of the predicate.

The mapping of logic predicates to Smalltalk messages is most easily explained by considering how we would write the invocation of a particular predicate as a Smalltalk message. As explained above, we use a naming convention for predicates where the name consists of a keyword for each argument the predicate takes, much like for method selectors in Smalltalk. When we want to invoke a predicate by sending a message to a logic module, we simply concatenate the keywords to get the selector of that message, without intervening colons except when we want to bind a value to a logic variable. The keywords for the last arguments are omitted if their corresponding parameters are not bound. In case that all arguments are left unbound, only the first keyword is used as a message selector (without colon).

For example, the logic predicate `add:with:to:/3` in the `Arithmetic` module defines the addition relation and it can be invoked in multiple ways. Therefore, the `Arithmetic` module understands the messages shown in table 1, where their corresponding query is also shown.

We further illustrate this with two example queries and their corresponding Smalltalk messages:

Calculate the addition of two numbers:

```
if Arithmetic.add:with:to:(1,2,?result)
    ?result -> 3
```

```
result := Arithmetic add: 1 with: 2.
```

Calculate the first argument of the addition, given the second argument and the result:

Message	Query
`add: 1 with: 2 to: 3`	`if add:with:to:(1,2,3)`
`add: 1 with: 2`	`if add:with:to:(1,2,?res)`
`add: 1`	`if add:with:to:(1,?y,?res)`
`add`	`if add:with:to:(?x,?y,?res)`
`addwith: 2`	`if add:with:to:(?x,2,?res)`
`addwithto: 3`	`if add:with:to:(?x,?y,3)`
`addwith: 2 to: 3`	`if add:with:to:(?x,2,3)`
`add: 1 withto: 3`	`if add:with:to(1,?y,3)`

Table 1. Mapping a multi-way predicate to messages

```
if add:with:to(?x,2,3)
    ?x -> 1
```

```
x := Arithmetic addwith:2 to:3.
```

A consequence of this approach is that for each message, we can document how many variables it returns (i.e. all arguments that were left unbound in the corresponding query of that particular message). The examples above use messages that return a single result for a single variable, but in table 1 many possible messages need to return more than one results for more than one variable.

Returning multiple variables When a message send leads to the invocation of a query that returns the binding of more than one variable, the results are returned in a Smalltalk `OrderedCollection` instance. This is not a break in our linguistic symbiosis, as we can document what the return result of that particular message is. For example the logic query to calculate the first two arguments of the addition that results in 3:

```
if add:with:to(?x,?y,3)
    ?x -> 1
    ?y -> 2
```

corresponds with the Smalltalk program:

```
| xyCollection |
xyCollection := Arithmetic addwithto:3.
```

In this example, two variables need to be returned, which means that a Smalltalk collection is returned containing the required `x` and `y` value. But we only showed one possible pair of results for `?x` and `?y`, while there are many more possible results (such as `?x -> 2 ,?y -> 1`).

Returning multiple results for each variable The returning of multiple results for each variable is a more complex part of the linguistic symbiosis. When a logic query also returns more than one result for each variable it returns, we can decide to either hide this from the Smalltalk programmer or explicitly return them in a collection. When returning all subsequent results as an explicit collection, the semantics are clear,

but the disadvantage is that the Smalltalk programmer most probably experiences that he is actually invoking a SOUL query here.
For example, consider the following Smalltalk program that invokes the `add:with:to:/3` logic predicate and prints the results for `x` on the Transcript.

```
xyColl := Arithmetic  addwithto: 3
xColl := xyColl first.
xColl do:[ :x | Transcript show: x]
```

Because the subsequent results for `x` are returned in an explicit collection, the Smalltalk program has to explicitly enumerate over the results. Another solution we have experimented with, is to hide the collection from the Smalltalk programmer and represent it as a single result. The return value appears as a single result to the programmer, but it actually represents all subsequent results. Internally, this means that we do use a kind of collection but when a message is sent to this 'implicit collection', the message is automatically dispatched to all values of the collection. For example, using the 'implicit collection' solution, we can write the above program as follows:

```
xyColl := Arithmetic addwithto: 3
x := xyColl first.
Transcript show: x
```

Mind that if the program above would be used with the 'explicit collection' solution, the collection `#(0 1 2 3)` will be printed to the Transcript. But, while the 'hidden collection' provides us with the desired result in many cases, it also leads to confusing and erroneous behaviour of Smalltalk programs since messages with side effects are executed multiple times. An improvement for this solution is to be researched. For now, we continue using the first solution with explicit collections.

4 Practical Use

In this section we present an example of a system that is implemented using both logic and OO programming. The example is that of an e-commerce system in which prices of products are adapted to take into account reductions granted to a specific user of the system. Which reductions are applicable typically depends on a number of things, such as the customer's history with the company, the use of e-coupons etc. Logic programming lends itself well to expressing the rules governing the applicability of reductions. In her master's thesis, Goderis describes an architecture for e-commerce systems that allows knowledge such as reduction rules to be described using logic programming while the rest of the system is implemented in OOP [1]. In this work she identified a need to be able to easily interchange information and control between SOUL and Smalltalk programs. The linguistic symbiosis mechanism we presented in this paper should fulfill this need.
For this example we will consider a simple e-commerce system with the two important classes `Product` and `Customer`. The message `price` can be used on an instance of `Product` to retrieve its price.

What we want to do now is apply changes to the return value of the `price` method to reflect price reductions granted to the current user of the system. Such an adaptation is typically done in Smalltalk using object wrappers. We have similarly defined a *logic module wrapper*, which is simply a special kind of logic module that can be used as a wrapper around an object. Any message sent to the wrapper is forwarded to the wrapped object unless the message is understood by the module in the sense that it defines a predicate that maps to the message.
The wrapper for `Product` instances we need here defines the predicate `price/1` as follows:

```
price(?reducedPrice) if
  ?wrapped.price(?basePrice),
  findall(?reduction, reduction(?reduction), ?reductions),
  reduced(?basePrice, ?reductions, ?reducedPrice)
```

The variable `?wrapped` is defined by the wrapper module as referring to the wrapped object. Here it is used to get the answer to the price message from the wrapped `Product` instance.
Several definitions for the reduction predicate can be given, we give a simple example here:

```
reduction(10) if
  customer(?customer),
  ?customer.age(?age),
  greaterThan(?age, 65)
```

The above rule expresses that a special "senior's reduction" of 10% is applicable to customers aged 65 or older.
Furthermore, the wrapping of instances should occur at run-time. The Smalltalk reflective facilities offer this functionality. The following Smalltalk method runs periodically and wraps products when they are eligible for reductions:

```
activateReductions
    allProducts do: [:product |  (knowledgeBase eligibleForReduction: product)
                                        ifTrue:[product wrap: ReductionWrapper new] ]
```

Using the symbiotic reflection and our linguistic symbiosis, it is now easy to describe when a product is actually eligible for reduction. The following logic rules do exactly this and are called by the Smalltalk program above.

```
eligibleForReduction:(?product,?result) if
    ?product.kind(toys),
    Date.today(?date),
    ?date.month(November).
```

The rule above describes that toys feature a price reduction in November.

5 Discussion and Future Work

In this paper, we describe ongoing work about the linguistic symbiosis of SOUL and Smalltalk. Because SOUL is a logic meta language over an object-oriented base language, this symbiosis requires a mapping of modules to classes and messages to queries. The advantage of

this symbiosis is that the base programmer can use the meta programs as if they were implemented in the base language. Furthermore, it also provides us with the opportunity to optimize parts of the logic meta programs by transparently replacing them with Smalltalk meta programs. Besides enhancing the practical use of symbiotic reflection [8], the linguistic symbiosis can also benefit non-reflective programming. Programs that do no reflective programming can also make use of the power of the logic programming language. As such, through our linguistic symbiosis, we have also introduced multi-paradigm programming in Smalltalk with the object-oriented and logic paradigms. We have already tackled many issues in this linguistic symbiosis, but others still remain open. We now summarize the most important results and elaborate on the open issues.

5.1 Results

Trivially, to obtain a linguistic symbiosis, a shared namespace for both languages should exist to allow access to the global entities of both languages. In our case, this namespace contains references to all classes and logic modules. Because these entities should look and feel the same in both languages, we need to define a mapping between them. In our particular case this means that we need to map predicates onto messages. This mapping has a syntactic aspect and a semantic aspect.
The syntactic mapping defines the common look, while the semantic mapping is concerned with the common feel. The syntactic mapping is rather language-specific, while the semantic mapping is paradigm-specific. Indeed, when using another object-oriented language (e.g. Java), the semantic mapping of one multi-way logic predicate to a set of methods will remain while a totally different syntactic mapping will be needed. Furthermore, we return multiple variables from a logic predicate in a container. The returning of multiple results for each of those variables can also correspond to returning them in an explicit container. Another solution we proposed is to hide this, but the implication of such a mapping is still to be researched.

5.2 Open issues

Multi-way methods? The linguistic symbiosis, as defined above, treats methods as 'uni-way' logic predicates. An issue that remains to be defined is how a method can correspond to a multi-way logic predicate. Clearly, this is not a trivial issue and it will most likely involve the implementation of a method for each way in which the predicate can be called. We can envision a system in which a group of methods that is implemented according to a pattern gives rise to a single predicate, much in the same way as a logic predicate is mapped onto multiple methods. On the other hand, there also exist a reasonable amount of logic predicates that are not multi-way. So maybe this trade-off would be acceptable.

Multiple Results? As we already mentioned, a logic query can return multiple results for multiple variables. These multiple results can

be hidden from the object-oriented programmer but this can lead to strange behaviour of object-oriented programs because messages will get executed several times.

Backtracking of side-effects? Even more complex issues arise when backtracking occurs in the logic program and methods that perform side effects have been executed. Of course, this is also true in pure logic programs that use logic predicates that execute side-effects.

Cross-language Inheritance? An issue that we did not mention at all, is how the inheritance relation could be implemented between logic modules and object-oriented classes. We consider this topic as future work.

5.3 Paradigm leaks

The above discussed problems of 'multiple results' and 'backtracking of side-effects' can be collectively described as 'paradigm leaks'. The symbiosis mechanism creates a leak where rules of one paradigm end up in the programming language based on the other paradigm. Dealing with and backtracking over multiple results in Smalltalk is not something considered to be part of the object-oriented paradigm, while side-effects are normally to be avoided in logic programming. Whether or not this leak is an undesirable effect in some cases remains to be investigated.

References

1. S. Goderis. Personalization in object-oriented systems. Master's thesis, Vrije Universiteit Brussel, 2001.
2. Y. Ichisugi, S. Matsuoka, and A. Yonezawa. Rbcl: A reflective object-oriented concurrent language without a run-time kernel. In *Proceedings of International Workshop on New Models for Software Architecture (IMSA): Reflection and Meta-Level Architecture*, pages 24–35, 1992.
3. K. Mens, I. Michiels, and R. Wuyts. Supporting software development through declaratively codified programming patterns. In *Proceedings of the 13th SEKE Conference*, pages 236–243. Knowledge Systems Institute, 2001.
4. T. Mens and T. Tourwe. A declarative evolution framework for object-oriented design patterns. In *Proceedings of Int. Conf. on Software Maintenance*. IEEE Computer Society Press, 2001.
5. W. D. Meuter. The story of the simplest mop in the world - or - the scheme of object-orientation. In *Prototype-based Programming*, pages 24–35. Springer-Verlag, 1998.
6. R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of TOOLS-USA '98*, 1998.

7. R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.
8. R. Wuyts and S. Ducasse. Symbiotic reflection between an object - oriented and a logic programming language. In ECOOP 2001 International workshop on MultiParadigm Programming with Object - Oriented Languages, 2001.