SOUL and Smalltalk - Just Married

Evolution of the Interaction Between a Logic and an Object-Oriented Language Towards Symbiosis

Kris Gybels*

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

1 Introduction

The *Smalltalk Open Unification Language* is a Prolog-like language embedded in the object-oriented language Smalltalk [5]. Over the years, it has been used as a research platform for applying logic programming to a variety of problems in object-oriented software engineering, some examples are: representing domain knowledge explicitly [3]; reasoning about object-oriented design [15,14]; checking and enforcing programming patterns [11]; ; checking architectural conformance [16] and making the crosscuts in Aspect-Oriented Programming more robust [6]. These examples fit in the wider research of *Declarative Meta Programming*, where SOUL is used as a meta language to reason about Smalltalk *code*.

Recently, we explored a different usage of SOUL in connecting business rules and core application functionality [2], which involves reasoning about Smalltalk *objects*. We found we had to improve on SOUL's existing mechanism for interacting with those objects because it was not transparent: it was clear from the SOUL code when rules were invoked and when messages were sent to objects, vice-versa solving queries from methods was rather clumsy. Ideally we would like to achieve a *linguistic symbiosis* between the two languages: the possibility for programs to call programs written in another language as if they were written in the same [8,13]. Such a transparent interaction would make it easy to selectively change the paradigm parts of an application are written in: if we find that a Smalltalk method is better written as a logic rule we should be able to replace it as such without having to change all messages invoking that method.

We will here take a historical approach to describing the SOUL/Smalltalk symbiosis. We would like to provide an insight into our motivation for and approach to achieve the symbiosis by contrasting three distinct stages in its evolution. In a first stage, SOUL was developed as a direct Prolog-derivate with some additional mechanisms for manipulating Smalltalk objects as Prolog values. In a second and third stage we explored alternative mechanisms and a more Smalltalk-fitting syntax for SOUL. Interestingly, when we performed a survey of other combinations of object-oriented and logic programming we found we could

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

easily categorize their approaches into one of our three "stages". The following sections discuss the stages in detail and the "Related Work" section at the end briefly discusses the survey.

2 Stage 1: Escaping from SOUL

The interaction mechanism found in the original SOUL can best be characterized as an escape mechanism. But before we go into this, let us make some general points about this version of SOUL:

Implementation: SOUL is embedded in Smalltalk, meaning it is entirely implemented in it.

Syntax: We assume readers are familiar with Prolog, the differences with this language and SOUL in this stage are:

- **Variable notation:** in Prolog, variables are written as names starting with a capital letter, in SOUL they are written as names preceded with a question mark, thus `Something` translates to `?something`.
- **List notation:** in Prolog, square brackets (`[ ]`) are used to write lists, these are replaced with angular brackets in SOUL (`< >`).
- **Rule notation:** the 'if' operator `:-` linking conclusion to conditions is replaced with the `if` operator in SOUL.

The combination of Smalltalk's Meta-Object Protocol and SOUL's embedding in Smalltalk lead to the insight that the simplest way to let SOUL programs reason about Smalltalk code is to give them access to the meta-objects directly. For this reason there are additional differences with Prolog:

Values: any Smalltalk object (not just the meta-objects) can be bound as a value to a logic variable.

Syntax: the Smalltalk term, a snippet of Smalltalk code enclosed in square brackets `[ ]`. The Smalltalk code can contain logic variables wherever Smalltalk variables are allowed.

Semantics: when Smalltalk terms are encountered as conditions in rules, they are "proven" by executing the Smalltalk code. The return value should be a boolean, which is interpreted as success or failure of the "proof". Smalltalk terms can also be used as arguments to conditions, then they are evaluated and the resulting value is used as the value of the argument. Unification deals with Smalltalk objects as follows: two references to an object unify only if they refer to the same object.

Primitive predicates: a primitive predicate `generate` can be used to generate elements of a Smalltalk collection as successive solutions for a variable.

The example set of rules in figure 1 are taken from SOUL's library for Declarative Meta Programming and show how Smalltalk terms are used. A predicate `class` is defined which reifies class meta-objects into SOUL; two different rules are defined for it to deal efficiently with different argument binding patterns. The

```
class(?x) if
  var(?x),
  generate(?x, [ System allClasses ])
class(?x) if
  nonvar(?x),
  [ ?x isClass ]

subclass(?super, ?sub) if
  class(?sub),
  equals(?super, [ ?sub superclass ])

hierarchy(?root, ?child) if
  subclass(?root, ?child)
hierarchy(?root, ?child) if
  subclass(?root, ?direct),
  hierarchy(?direct, ?child)
```

Fig. 1. Example rules defining predicates for reasoning about Smalltalk programs.

```
argumentArray := Array with: (Array with: #x with: someClass).
evaluator := SOULEvaluator eval: 'if hierarchy(?x, ?y)' withArgs: argumentArray.
results := evaluator allResults.
ysolutions := results bindingsForVariableNamed: #y.
```

Fig. 2. Code illustrating how the SOUL evaluator is called from Smalltalk and how the results are retrieved.

`subclass` predicate expresses that two classes are related by a direct subclassing relationship when one is the answer to the `subclass` message sent to the other. The `hierarchy` predicate extends this to indirect subclassing relationships.

The example rules are indicative for the way SOUL interacts with Smalltalk in this stage: the use of Smalltalk terms is limited to a small collection of predicates such as `class` and `subclass`, which are organized in the so-called "basic layer". Other, more high-level predicates such as `hierarchy` make use of the predicates in the basic layer to interact with Smalltalk objects. This organization avoids pollution of the higher-layer predicates with explicit escape code[1]. In a way, the basic layer provides a gateway between the two languages by translating messages to predicates and vice-versa.

The other direction of interaction, from Smalltalk to SOUL, is done through explicit calling of the SOUL evaluator with the query to be evaluated passed as a string. Figure 2 illustrates how the `hierarchy` predicate is to be called.

[1] Another reason why this is done is to make the higher-layer predicates less dependent on Smalltalk, so that they may later be used when reasoning about code in other OO languages [4].

On the second line, an evaluator object is created by sending the message `eval:withArgs:` to the `SOULEvaluator` class, the message is passed the query to evaluate and variable bindings as arguments. The variable bindings are passed as an array of variable-object pairs. In the example, the logic variable `?x` will be bound to the value of the Smalltalk variable `someclass`, so the query will search for all child classes of that class. These child classes will then be bound as solutions to the variable `?y`. These solutions can be retrieved by sending an `allResults` message to the evaluator object, which returns a result object. The result object then needs to be sent the message `bindingsForVariableNamed` to actually retrieve the bindings, which are returned as a collection.

3 Stage 2: Predicates as Messages

A second stage of SOUL-Smalltalk interaction, which we reported on at a previous multi-paradigm programming workshop [1], aimed at providing more of a transparent interaction. Our motivation then was especially to improve on the way Smalltalk programs can invoke queries, and do it in a way that would provide *linguistic symbiosis*. To do so, we tried to map invocation of predicates more directly to the concept of sending a message.

The term linguistic symbiosis refers to the ability for programs to call programs written in another language as if they were written in the same. Having this ability would also imply that transparent replacement is possible: replacing a ”procedure” (= procedure/function/method/...) in the one language with a ”procedure” in the other, without having to change the other parts of the program that make use of that ”procedure”. In fact, the term was coined in the work of Ichisugi et al. on an interpreter written in C++ which could have all of its parts replaced with parts written in the language it interprets. Such usage of linguistic symbiosis to provide reflection was further explored in the work of Steyaert [13].

While these earlier works provided us with solutions, we also had an added problem: the earlier works dealt with combining two languages founded on the object-oriented paradigm, while we aimed at combining an object-oriented and a logic language. The earlier works dealt with mapping a message in the one language to a message in the other, while we needed to map messages to queries.

To provide a mapping of messages and queries, we had five issues to resolve:

Unbound variables: how does one specify in such a message that some arguments are to be left unbound? The concept of 'unbound variables' is foreign to Smalltalk.

Predicate name: how is the name of the predicate to invoke derived from the name of the message?

Returning multiple variables: how will the solutions be returned when there are multiple variables in the query?

Returning multiple bindings: if there are multiple solutions for a variable, how will these be returned?

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

Table 1. Mapping a predicate to messages

Receiver: which object will the message be sent to?

We combined the solution for the first two issues by assuming that predicate names, like Smalltalk messages, would be composed of keywords, one for each argument. To specify which variables to leave unbound we adopted a scheme for combining these keywords into a message name from which that specification can be derived. To invoke a predicate from Smalltalk one would write the message as: the name of the first keyword, optionally followed by a colon if the first argument is to be bound and a Smalltalk expression for the argument's value, then the second keyword, concatenated to the first if that one was not followed by a colon, and again itself followed by a colon if needed for an argument and so on for the other keywords until no more keywords need to follow which take an argument. This is best illustrated with an example. Table 1 shows the 2^3 ways of invoking a predicate called `add:with:to:` and the equivalent query in SOUL.

For the issue of needing a receiver object for the message, we mapped layers to objects stored in global variables. Because in Smalltalk classes are also objects stored in global variables, this has the effect of making a predicate-invoking message seem like a message to a class. The basic layer is for example stored in `Basic`.

We proposed two alternative solutions to the issues of returning bindings. The first was simply to return as result of the message a collection of collections: a collection containing for each variable a collection of all the bindings for that variable. The alternative consisted of returning a collection of message forwarding objects, one for each variable. Sending a message to such a forwarding object would make it send the same message to all the objects bound to the variable. The idea was to provide an implicit mechanism for iterating over all the solutions of a variable, very much how like SOUL can backtrack to loop over all the solutions for a condition. This however lead to matters such as whether forwarding objects should also start backtracking over solutions etc., so it was discarded as a viable solution. We coined the term *paradigm leak* to refer to this problem of concepts "leaking" from one paradigm to the other.

We also used the predicate and message mapping to replace SOUL's earlier use of Smalltalk terms. Instead of using square brackets to escape to Smalltalk for sending a message, the same message can now be written more implicitly as an invocation of a predicate in an object "pretending to be a SOUL module". Here, the reverse of the above translation happens: SOUL will transform the predicate to a Smalltalk message by associating the arguments of the predicate to the keywords in its name. The predicate's last argument will be unified with the result of the actual message send. Take the following example:

```
if Array.with:with:with:(10,20,30, ?instance), ?instance.at:(2,?value)
```

The first condition in the example query will actually be evaluated by sending the message `with: 10 with: 20 with: 30` to the class `Array`. The result of that message is a new `Array` instance, which will be bound to the variable `?instance`. In the second condition, the message `at: 2` will be sent to the instance and the result, 20 in this case, will be bound to the variable `?value`.

While in this second-stage SOUL mixing methods and rules is entirely transparent from a technical standpoint, it is obvious which code is intended to invoke what to a human interpreter. Technically there is no more need in SOUL for an escape mechanism, and the same language construct is used to invoke rules and messages. Similarly in Smalltalk, queries no longer have to be put into strings to let them escape to SOUL and can just be written as message sends. However, a Smalltalk programmer would frown when seeing messages such as `addwith: 2 to: 3`. Furthermore, he would probably guess that the result of that message would be the value 5, instead it will be a collection with a collection containing the value 5. The keyword-concatenated predicate names in SOUL also lead to awkward looking programs in that language.

4 Stage 3: Linguistic Symbiosis?

The next, and currently last, stage in the SOUL-Smalltalk symbiosis uses a new syntax for SOUL to avoid the clumsy name mappings from the previous stage. For this stage we also had a specific application for the symbiosis in mind, business rules [2], which influenced its development in certain respects. One difference is that previously we wanted to allow Smalltalk programs to call the existing library of SOUL code-reasoning predicates, while for supporting business applications the idea is rather to use SOUL to write new rules implementing so-called business rules of the application. This also implies another shift: reasoning about (business) objects rather than meta objects.

In the new syntax predicates look like message sends. Let us illustrate with an example, figure 3 contrasts the classic `member` predicate with its new `contains:` counterpart.

The second rule for `contains:` can be read declaratively simply in Prolog-style as "for all `?x`, `?y` and `?rest` the `contains:` predicate over `<?y | ?rest>` and `?x` holds if ....". A declarative message-like interpretation could read "for all `?x`, the answer to the message `contains: ?x` of objects matching `<?y | ?rest>` is true if the answer of the object `?rest` to `contains: ?x` is true." Both interpretations are equivalent, though the second one is really the basis for the new symbiosis.

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

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

Fig. 3. Comparison of list-containment predicate in classic and new SOUL syntax.

```
?product discountFor: ?customer = 10 if
  ?customer loyaltyRating = ?rating &
  ?rating isHighRating
```

Fig. 4. Example of a rule using the equality operator.

Because messages can return values other than booleans, we added another syntactic element to SOUL to translate this concept to logic programming. The equality sign is used to explicitly state that "the answer to the message on the left hand side of `=` is the value on the right hand side". Figure 4 shows an example.

The new syntax has a two-fold impact on how the switching between Smalltalk and SOUL occurs. It is no longer necessary to employ a complicated scheme with concatenation of keywords to get the name of a predicate. Another is that there is no more mapping of objects to SOUL modules and vice-versa, modules were dropped from SOUL as the concept of having a "receiver" for a predicate now comes as part of the message syntax.

A Smalltalk program no longer has to send a message to a SOUL module "pretending to be an object" to invoke a query. Instead, a switch between the two languages now occurs as an effect of method and rule lookup: we changed Smalltalk so that when a message is sent to an object and that object has no method for it, the message is translated to a query. In SOUL, when a rule is not found for the predicate of a condition, the condition is translated to a message. This new scheme makes it much easier and much more transparent to actually interchange methods and rules.

The translation of queries and messages is straightforward and we'll simply illustrate with another example. Figure 5 shows a price calculation method on a class `Purchase` which loops through all products a customer bought and sums up their total price minus a certain discount. When the `discountFor: customer` message is sent to the products, Smalltalk will find no method for that message, so it will be translated to the query:

```
if ?arg1 discountFor: ?arg2 = ?result
```

```
Purchase instanceVariables: 'shoppingBasket customer'

Purchase>>totalPrice

  | totalPrice discountFactor |

  totalPrice := 0.
  shoppingBasketContents do: [ :aProduct |
    discountedFactor := (100 - (aProduct discountFor: customer)) / 100
    totalPrice := totalPrice + (discountFactor * aProduct price)
  ]
```

Fig. 5. Example price calculation method on Purchase class

Where `?arg1` and `?arg2` are already bound to the objects that were passed as arguments to the message. When the query is finished, the object in `?result` is returned as result of the "message". This returning of results is actually a bit more involved, we'll discuss it further in the next section.

For the inverse interaction, we can take the `loyaltyRating =` condition in the `discountFor:` rule (fig. 4) as an example. For a small business the loyalty rating of a customer can simply be stored as a property of the customer object which can be accessed through the `loyaltyRating` message. In that case, SOUL will find no rule for the "predicate" `loyaltyRating` and will translate the condition simply to the message `loyaltyRating` which is then sent to the customer object in the variable `?customer`. After it returns, the result of the message is unified with the variable `?rating`. Of course, for a bigger business we might want to replace the calculation of `loyaltyRating` with a set of more involved business rules which we'd prefer to implement with logic programming, for example "a high rating is given to a customer when she has already spent a lot in the past few months". With the transparent symbiosis such a replacement is easy to do.

5 Limits and Issues

At the end of the "Stage 2" section, we remarked that our solution then was only technically transparent, it was rather obvious to a programmer which code was intended to invoke which paradigm. In the previous section we demonstrated that this is now much less the case, it is fairly easy now to interchange methods and rules without this becoming obvious. There are however limits to this interchanging and there are still subtle hints that may reveal what paradigm is invoked. These limits and issues stem from differences in programming style between the object-oriented and logic paradigms.

One important style difference between the paradigms is the way multiplicity is dealt with. In logic programming, there is no difference between using a predicate that has only one solution and one that has multiple solutions. In

```
?child ancestor = ?parent if
  ?child parent = ?parent.
?person ancestor = ?ancestor if
  ?person parent = ?parent,
  ?parent ancestor = ?ancestor
```

Fig. 6. Rules expressing the ancestor relationship between Persons

```
Person instanceVariables: 'name parent'

Person>>parent
   ^ parent

Person>>name
  ^ name

Person>>printOn: stream

  name , ' descendant of ' printOn: stream.
  self ancestor do: [ :ancestor |
     ancestor name , ' and ' printOn: stream
  ]
```

Fig. 7. Instance variables and some methods of the Person class

object-oriented programming there is an explicit difference between having a message return a single object or a collection of objects (even, or especially, if there's only one object in that collection). This difference leads to an issue in how results are returned from queries to Smalltalk, and one in how predicates and messages are named.

When a Smalltalk message invokes a SOUL query and the query has only one solution, should the solution object be simply returned or should a singleton collection with that object be returned? The invoking method may expect a collection of objects, which would then just happen to contain just a single item, or it may generally be expecting there to be only one result. It is difficult for SOUL however to know which is the case. To deal with this we made SOUL return single solutions in a `FakeSingleItemCollection` wrapper. The `FakeSingleItemCollection` class implements most of the messages expected of collections in Smalltalk, any other messages are forwarded to the object that is being wrapped. There is thus an "automatic adaptation" to the expectations of the invoking method.

Plurality, or lack thereof, in the names of predicates and messages can cause some programming style difficulties. Figures 6 and 7 illustrate the modeling of persons and their ancestral relations through a class and some logic rules. Invoking these rules from the `printOn:` method is however awkward: it is quite natural for a logic programmer to write the relationship as ”ancestor” even though there will be multiple ancestors for each Person, the object-oriented programmer would however prefer to write the plural ”ancestors” to indicate that a collection of results is expected. One solution to this problem is to implement a rule for `ancestors` which simply maps to `ancestor`, this would however defeat the purpose of having an automatic mapping of messages and queries. A potential solution could be to take this style difference into account when doing the mapping by adding or removing the suffix *-s* when needed.

When comparing the stage 2 and stage 3 symbiosis, stage 3 may seem more limited in the variables that can be left unbound when invoking queries from Smalltalk. In stage 2 the mapping of predicate names to message names implicitly also indicated which variables to leave unbound, while in stage 3 the mapping of messages to queries only leaves unbound the result variable, the one on the right hand side of the equality sign in the query. Actually, we did implement a means for leaving other variables unbound as well. We changed the way Smalltalk deals with temporary variables to allow for the following code to be written:

```
| products customers discounts |

discounts := products discountFor: customers
```

Normally the Smalltalk development environment would warn that this code uses temporary variables before they are assigned. Now however, the message `products discountFor: customers` will result in the query:

```
if ?arg1 discountFor: ?arg2 = ?result
```

Where all of `?arg1`, `?arg2` and `?result` are left unbound. When the query is finished, the result of the message will be as described earlier and additionally the temporary variables `products` and `customers` will also be assigned the solutions of the variables `?arg1` and `?arg2`.

This leaving unbound of temporary variables is however another example of a paradigm leak, it is quite unnatural code for a Smalltalk programmer to write. We consider it as something that should be used with care and preferably avoided. While in stage 2 the equivalent mechanism seemed most necessary because the motivation was to allow access to all *existing* SOUL predicates, our focus shift to implementing *new* business rules makes it less necessary: its better to design the rules differently. Nevertheless many of the rules will be designed to be used from other rules, not to be replaced with methods and callable in a multi-way fashion. On occasion, these rules may need to be used directly from a method, so we kept the unbound temporaries mechanism in place.

There is also a limitation in leaving arguments unbound the other way around: when translating a condition to a message, all of its arguments are expected to be bound. SOUL will currently generate an error otherwise. It would be possible though to at least deal with the ”receiver” argument of the condition in a more logic-like way: when it is unbound, SOUL could send the message to some random object from memory which support a method for the message. If the message's result is true, the object is a solution for the "receiver" variable. On backtracking all of the other objects supporting the message would be tried. A problem here would be the accidental invocation of object-mutating messages due to polymorphism: when posing the query `?game draw` we may simply be interested in all chess games that ended in a draw but may wind up also drawing all graphical objects on the screen. In practice though this may not be so much of a problem as normally the messages invoked from SOUL would have a keyword "is" or "has" in their name because they are written as invocation of predicates, and it is a convention normally applied by Smalltalk programmers as well.

6 Related work

We examined several existing systems which were designed or could be used for business rule development and in which object-oriented and logic programming are combined [2]. The interaction mechanisms we encountered fit in one of three categories similar to the three stages we discussed here: use of an escape mechanism, some explicit mapping of predicates and methods or a syntactic and semantic integration of the two languages. We limit our discussion here mostly to a few systems that aim for the third category as well.

NéOpus also extends Smalltalk with logic programming, though with production-rule based logic rather than proof-based logic [12]. Rules consist of conditions and actions, rather than conclusions, which are respectively expressed as boolean messages to objects and state-changing messages to objects. The concept of a "conclusion" as something separate from a direct effect on the state of objects is thus dropped. Rules are also not invoked through queries, but rather are triggered by changes in the state of objects and there is no backtracking to generate multiple solutions. This means that some of the issues we had to deal with do not occur in NéOpus: the problems of mapping predicates and methods, returning of multiple results etc. Pachet in fact argues against adding backward chained inferencing to NéOpus because he finds there's a contradiction between the desire to use the OO language to express rules in and allow backward chaining [12], which may come down to our issues. Note that we made the rule language resemble the object-oriented one as closely as possible and needed to allow for symbiosis, we did not simply use the OO language directly to express rules. Besides what form of chaining to use, Pachet also discusses other questions which we had to resolve as well. Most importantly what happens to pattern matching and object encapsulation. Often in logic programming a data structure is accessed directly through unification of its constituent parts. In some of the other systems we examined, like CommonRules [7] , this is still done this way by mapping objects to predicates with an argument for each instance variable. In SOUL, as in NéOpus, we chose to uphold object encapsulation and only allow accessing objects through message sending.

LaLonde and Van Gulik used Smalltalk's reflection to turn ordinary methods into backtracking methods [10]. They built a small framework[2] to support the backtracking methods. Most important in there is a message which makes its calling method return with a certain value but remembers the calling point, the method can then be made to resume execution from that point on. This is achieved by exploiting Smalltalk's ability to access the execution stack from any method. The backtracking takes care of undoing changes to local variables, though not to instance variables and globals . Local variables are thus used to simulate logic variables, but they are assigned rather than unified and there is no simulation like our unbound temporaries for calling methods with some arguments left unbound, so backtracking methods are no full simulation of logic rules. Despite the similarities in the use of Smalltalk expressions, programming in this system seems quite different from programming in symbiotic SOUL.

Kiev [9] extends Java with logic rules, which can be added directly to classes and called through message sending. To call a rule with unbound arguments, one passes an empty wrapper object as argument which will then be bound by the rule. A new for-construct can be used to iterate over all solutions. There is no equivalent for our equality operator construct, calling a rule from a method as a message always returns a boolean to indicate success or failure. This is a subtle but important difference with symbiotic SOUL: returning objects from rules requires the use of sending a message with unbound arguments, making calling rules not as transparent.

7 Summary and Conclusions

We presented the history of a combination of Smalltalk with a logic language. Three distinct stages appeared in its evolution of the interaction between the two languages which we also encountered in studying other combinations of object-oriented and logic programming: a stage where the languages could bind each other's values to variables and manipulate these values by "escaping" to the other language, a stage where the escape mechanism was made more transparent by an automatic mapping of predicates and methods and the current final stage in which the syntax of the logic language has been adapted to that of the host language to allow not only for technical but programming style transparency as well. The aim was to achieve a linguistic symbiosis so that methods and rules can be easily and transparently interchanged. This is not just of theoretical interest but has an application in the development of business rule applications: an existing application without business rule separation may need to be turned into one that does, or new developments in the policies of the business may make it more interesting to turn methods into rules.

We compared our earlier and current solution for such issues as how to map messages and queries, return multiple results from a query to Smalltalk etc. There are unfortunately still some minor issues to resolve such as how to

[2] Small enough to have the full code listed in their paper.

deal properly with the difference in use of plurality in names between the two paradigms and avoiding the invocation of state-changing messages. Nevertheless we have found the current version of symbiotic SOUL to be a great improvement over previous versions.

References

1. Johan Brichau, Kris Gybels, and Roel Wuyts. Towards a linguistic symbiosis of an object-oriented and logic programming language. In Jörg Striegnitz, Kei Davis, and Yannis Smaragdakis, editors, *Proceedings of the Workshop on Multiparadigm Programming with Object-Oriented Languages*, 2002.
2. Maja D'Hondt and Kris Gybels. Linguistic symbiosis for the automatic connection of business rules and object-oriented application functionality. (to appear), 2003.
3. Maja D'Hondt, Wolfgang De Meuter, and Roel Wuyts. Using reflective logic programming to describe domain knowledge as an aspect. In *First Symposium on Generative and Component-Based Software Engineering*, 1999.
4. Johan Fabry and Tom Mens. Language-independent detection of object-oriented design patterns. In *Proceedings of the European Smalltalk User Group's conference*, 2003. (Conditionally accepted).
5. Adele Goldberg and Dave Robson. *Smalltalk-80: the language.* Addison-Wesley, 1983.
6. Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of the Second International Conference of Aspect-Oriented Software Development*, 2003.
7. IBM. Business rules for electronic commerce: Project at IBM T.J. Watson research, 1999. http://www.research.ibm.com/rules/.
8. 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.
9. Maxim Kizub. *Kiev language specification*, July 1998. http://www.forestro.com/kiev/kiev.html.
10. Wilf R. LaLonde and Mark Van Gulik. Building a backtracking facility for Smalltalk without kernel support. In *Proceedings of the conference on Object-Oriented Languages, Systems and Applications.* ACM Press, 1988.
11. Kim Mens, Isabel Michiels, and Roel Wuyts. Supporting software development through declaratively codified programming patterns. In *Proceedings of the 13th SEKE Conference*, 2001.
12. Francois Pachet. On the embeddability of production rules in object-oriented systems. *Journal of Object-Oriented Programming*, 8(4), 1995.
13. Patrick Steyaert. *Open Design of Object-Oriented Languages.* PhD thesis, Vrije Universiteit Brussel, 1994.
14. Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of TOOLS-USA 1998*, 1998.
15. Roel Wuyts. *A Logic Meta Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation.* PhD thesis, Vrije Universiteit Brussel, 2001.
16. Roel Wuyts and Kim Mens. Declaratively codifying software architectures using virtual software classifications. In *Proceedings of TOOLS-Europe 1999*, 1999.