Building Software with Logic and OO Symbiosis: an Experience Report

Johan Fabry*
Johan.Fabry@vub.ac.be

Kris Gybels†
Kris.Gybels@vub.ac.be

May 19, 2004

Abstract

In this paper we present the results of a case study of using multi-paradigm programming, more concretely, Logic and OO symbiosis. The case study consists of the implementation of sections of a weaver for aspect-oriented programming. We have implemented such a weaver for transaction management, and in this paper we show how multi-paradigm programming in the logic and object-oriented paradigm using language symbiosis significantly aids implementation. First we give an overview of the linguistic symbiosis provided by our logic language SOUL and the object-oriented language Smalltalk. Secondly, we detail how we used the symbiosis to implement parts of the weaver, and thirdly, we discuss the advantages and shortcomings of the approach.

1 Introduction

The goal of Aspect-Oriented Programming (AOP) [10] is to modularize crosscutting concerns. Crosscutting concerns are concerns or features of a program that cut across the program, meaning they are not localized in a specific module. The AOP observation is that certain concerns are inherently crosscutting in programs that rely on the modularization mechanisms that are currently in use such as procedures, modules and classes because they are all based on the idea of one module calling another. Most AOP research has therefore concentrated on providing new modularization mechanisms where so-called aspects can themselves specify how they crosscut other modules and exactly where or at what points they do so. Often-cited examples of crosscutting concerns are logging, persistence, synchronization and transaction management [11, 12, 13].

Aspect languages are programming languages that provide these new modularization mechanisms, typically as an extension of an existing language. An aspect language usually defines two conceptual entities: the pointcuts and the advice. Pointcuts define where in the execution of the code (so-called join points) the aspect should be invoked, and advice specifies the body of the aspect, which will be invoked.

*Author funded by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT) in the context of the CoDAMoS project.

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

An aspect weaver is used to combine the semantics of aspects with the regular modules, which can be done in several different ways. However, because of the development of aspect languages as extensions to existing languages, this is most often done by literally weaving the code of the aspects into the code of the regular modules. This produces a program in the original language which can be compiled with an existing compiler for that language.

Logic languages in general and in particular the languages TyRuBa and SOUL have been demonstrated to be excellent vehicles for aspect specification and source-code-based weaving [3, 4, 9]. Both languages are variants of Prolog used for *logic meta-programming*: the use of a logic language for reasoning about programs.

In this paper we discuss how a change in the interaction mechanism of SOUL [14] with the underlying Smalltalk system impacts an existing aspect weaver which makes use of SOUL. This aspect weaver [6] was developed in parallel with SOUL. Different parts of the weaver are indeed written both in Smalltalk and SOUL, but originally used a primitive interaction mechanism. As SOUL evolved, the SOUL language has been changed to allow for a more transparent interaction between SOUL and Smalltalk code, known as *linguistic symbiosis* [8]. We have re-implemented the parts of the weaver which used this interaction mechanism to use logic OO symbiosis, and we report our results here.

The paper is structured as follows: In the next section, we first briefly discuss the actual goal of the aspect weaver: allowing the specification and weaving of transaction management aspects. In section three we introduce the concepts of Logic Meta Programming and language symbiosis, and how this is implemented in SOUL. Section four details the case in more detail, providing three instances of logic and OO symbiosis. A discussion is presented in section five, and section six concludes.

2 Transaction Management as an Aspect

Before talking about the logic smalltalk symbiosis, we first provide some context and consider transaction management as an aspect.

Transactions are a widely used construct in distributed systems to manage concurrency and failures. A transaction is a sequence of program instructions that are to be considered as an indivisible block. Such blocks, when executed concurrently, may not interfere with each other's data and must maintain global data consistency. Transactions are widely supported in distributed system infrastructure tools and are a de facto standard for concurrency management when using databases.

When writing an application which uses transactions the transaction programmer will mark the start and end of each transaction in the application code, so-called transaction demarcation, which will associate each data access with a running transaction. At run-time, transaction monitor software will mediate concurrent accesses to the data by the different transactions within the application.

It is obvious that such transaction demarcation code will be spread out throughout a large part of the application, i.e. everywhere shared data is used. Therefore we can consider the management of transactions as a crosscutting concern, and it is beneficial to define this as an aspect. Indeed, there already

exists a body of work to define transaction management as an AspectJ [1] aspect, either stand-alone by Kienzle and Gerraoui [11] or as a by-product of specifying a persistence aspect by Soares et al.[13] and also by Rashid and Chitchyan [12].

The long-term goal of our transaction management research is enhancing transaction management through the use of high-level semantic information. In this context we wish to use aspect-oriented programming to specify such advanced transaction management. When evaluating the existing work, we found that the above approaches were insufficient for our needs, and therefore we had to provide our own implementation of transaction management as an aspect.

Because of the extensive support for distributed systems in the J2EE standard, especially in the Enterprise JavaBeans architecture, we have chosen Enterprise JavaBeans as a target domain for our weaver. In other words, our aspect weaver will weave the transaction aspect into Java code that is compliant with the Enterprise JavaBeans architecture. We will not detail this architecture here, as it is outside of the scope of this paper.

With our weaver, transaction boundaries coincide with method boundaries: transactions can only be started up as a method begins, and transactions stop at the end of a method. This implies that to specify transaction boundaries, the application programmer simply needs to specify a method. This is done in a separate aspect file, using an aspect-specific language. Aspect programs in this language specify a class and list a number of method signatures, postfixed either with `new` or `none`, indicating whether a new transaction should be started or the method is not transactional. For the parameter list of the method signature, the `*` wildcard may be used, indicating applicability regardless of parameter types. Also, default behavior for a bean can be set to be either `new` or `none`. Lastly, the programmer can define exception handlers for methods, by simply appending a number of `catch` blocks, containing java code, to the transactional declaration of the method.

An example aspect definition containing these kinds of exception handlers is given below:

```
transactions CounterBean
{
   increment(int count) new
      catch (OverflowException ex)
         {this.enlarge(count); this.increment(count);};
   get(*) none;
   default new
      catch(Exception ex)
         { txRollback(); ex.printStackTrace(); System.exit(1)};
}
```

At weaving time, the aspect weaver will use such a definition, to determine what the transactional properties for each method are. Note that the different ways a method can be matched with its transactional properties can interact in a number of ways, and the weaver needs to take care of this. We will show in section 4 how this is implemented using SOUL and its linguistic symbiosis feature. However, we first need to discuss Logic Meta Programming and the linguistic symbiosis, which we will do next.

3 Logic Meta Programming and Linguistic Symbiosis

SOUL [14] is basically a variant of Prolog which was implemented in Smalltalk. Besides some minor syntactical differences, SOUL also differs from Prolog in that it allows interaction with the underlying Smalltalk system. Smalltalk objects can be contained in logic variables and can be sent messages from within SOUL. Originally, this message sending was achieved by allowing the use of embedded Smalltalk expressions in logic rules, in newer versions of SOUL this was changed to a more transparent form of linguistic symbiosis. We will explain these differences further on, but will first illustrate how such interaction is actually used in LiCoR and Irish.

3.1 LiCoR

Because SOUL is used for logic meta-programming, a whole library of predicates for reasoning about code has been developed. This is the Library for Code Reasoning (LiCoR). The library is intended to be as language independent as possible, so that it can be used not only to reason about Smalltalk programs but also about programs in other object-oriented languages. To this end, the predicates in the library are organized into several different layers, where the lower-most layer provides predicates to reason about the basic elements of code such as classes and their methods, and their basic relationships such as one class being the subclass of another. Higher-leveled layers provide predicates for ever more complex relationships such as classes being in each other's hierarchy, classes implementing a visitor design patterns [7] etc. This allows the upper layers to be almost completely language-independent, while the lowest layer is the only language-dependent layer [6].

For Smalltalk, we can use Smalltalk's built-in reflection, which easily gives us a representation of classes and their methods as objects. The lower-most LiCoR layer uses the Smalltalk-SOUL interaction mechanism to directly make use of the Smalltalk reflection objects, which provides a simple reification into SOUL. Consider for example the following rule:

```
class(?c)  if  memberOfCollection(?c, [System allClasses]).
```

This rule expresses that the `class` predicate is true for any object that is in the collection as returned by a message `allClasses` to the Smalltalk `System` object. The predicate `memberOfCollection` is similar to the `member` predicate in regular Prolog, but works on Smalltalk collections instead of Prolog lists. The interaction mechanism used here is that of the older version of SOUL where a snippet of Smalltalk code can be used as a logic term. When Smalltalk terms are used as arguments to predicates, their code is executed during the unification process and the resulting Smalltalk object is used for the actual unification. When used as a condition in a rule, the code is executed as a way of "proving" the condition and is expected to return a boolean value which is interpreted as success or failure of the "proof". In the `class` example rule a Smalltalk term is used as an argument to the `memberOfCollection` predicate, the actual collection returned by the term will then be unified with the second argument of a rule for `memberOfCollection`.

3.2 Irish

To be able to use SOUL for the specification of aspects for Java programs, Irish, an extension of LiCoR for Java was developed. Irish provides a different basic layer for LiCoR which uses a Java parser, written in Smalltalk, to provide a Smalltalk representation of Java programs.

As we have said above, in section 4 the transaction aspect weaver for Java will be discussed. Since its weaving involves reasoning about the bodies of Java methods, it is of particular relevance to us here how method bodies are represented in Irish. Method bodies are fully reified as functors, i.e. all statements of the method and all the expressions contained within these statements, up to, and including their references to variables or literals are reified. This representation is much more suitable for writing rules that reason about the contents of methods than a representation that would just use plain strings. An example is shown below where we have the method of a `Counter` class in an `example` package, first as plain Java, and secondly in its functor representation used in Irish. This representation is best read as a tree, of which the nodes are tagged, for example by `method` or `variable`.

```
public String toString(){return Integer.toString(count);}

method(example.Counter
   signature(<public>,<java.lang.String,0>,toString,<>),
   arguments(<>), temporaries(<>),
   statements(<return( send(java.lang.String, 0,
        variable(java.lang.Integer, 0, Integer),
     toString,
     <variable(int, 0, count)>))>))
```

One can see that this representation is quite extensive, also specifying the types and dimensions of each expression within the method. For brevity, in the remainder of this text, we use the name *sub-method entities* to refer to the statements, expressions, variable references and literals within the method body.

If we consider such a sub-method entity, for example the message send `send(...)` included above, we can write a simple logic rule that matches each instance of that kind of entity, in our example, all message sends. In other words, the logic rule `isMessageSend( send( ?rtype, ?rdims, ?receiver, ?message, ?arguments))`, when called with as argument the send datastructure given above, will return `true`. Given any other kind of argument, the rule will fail, i.e. return `false`. Furthermore, we can easily obtain the return type of the message send above using the following rule[1]: `messageSendRType( send( ?rtype, ?, ?, ?, ?), ?rtype)` If the data-structure of the message send given above is passed as the first argument, the logic interpreter will yield, in the second argument, `java.lang.String` as the return type for the message send.

The above two interactions with the logic engine: identifying if a data structure corresponds to a given parse tree node, and returning a (part of) this node, are used by LiCoR to implement a parse tree traversal. This traversal takes two

[1] '?' indicates an anonymous variable, it is the SOUL equivalent of Prolog's underscore variable.

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

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

Figure 1: Comparison of list-containment predicate in classic and new SOUL syntax.

main arguments: a `?found` logic rule[2] which matches the parse tree element to be found, and a `?process` logic rule that processes this element, to return the required sub-element. The traversal will then recursively travel through the logic parse tree, at each node trying the match rule. If the match rule succeeds, the process rule will be fired on that node, and if not, the traversal will recurse through all sub-nodes of that node. The signature of the parse tree traversal rule is as follows: `traverseMethodParseTree( ?body, ?env, ?found, ?process)`. The `?body` argument contains the method body to be traversed and the `?env` argument is used to return the results of the process rule.

3.3 Linguistic Symbiosis

In the previous sections we briefly introduced an older version of SOUL and its Smalltalk term interaction mechanism. More recently this mechanism was replaced with a different one with the goal of having a more *transparent linguistic symbiosis*. The mechanism is linguistically transparent in the sense that a distinct syntactic construct such as the Smalltalk term is no longer needed. Instead the interaction mechanism is absorbed into the rule and method lookup of SOUL and Smalltalk respectively. For this to be possible, the syntax of SOUL was changed to more closely match that of Smalltalk. Instead of the classic Prolog syntax, a Smalltalk-message-like syntax is now used for functors. Figure 1 illustrates the use of both syntaxes for implementing a list-containment predicate. The second rule for `contains:` can, on the one hand, be read in Prolog-style as "the `contains:` predicate over `<?y | ?rest>` and `?x` holds if ...". On the other hand, a declarative message-like interpretation would be "for all `?x`, the answer to the message `contains: ?x` , sent to objects matching `<?y | ?rest>`, is true if the answer of the object `?rest` to the message `contains: ?x` is also true." Both interpretations are equivalent, though the second one is really the basis for the new symbiosis.

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 Smalltalk message on the left hand side of `=` is the value on the right hand side". This is used in the example in figure 2 which is discussed below.

To actually allow SOUL-Smalltalk interaction the mechanism for rule lookup

[2]Logic variables may also be bound to logic rules.

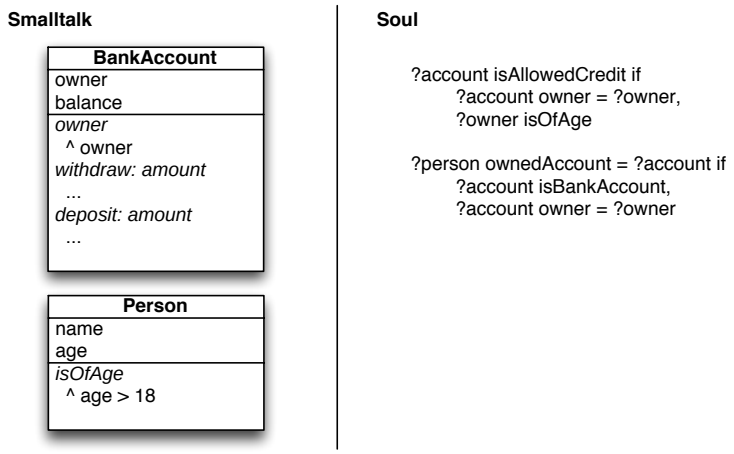

Figure 2: Illustration of code that would involve translation of messages and conditions.

was changed. When a rule is not found for a condition that is to be proven, SOUL will translate the condition to a Smalltalk message instead. This translation is particularly straightforward because of the Smalltalk-message-based syntax for conditions. One way of understanding this is that "the square brackets of Smalltalk terms are automatically put around a condition when no rule to prove it with is found". A simple example to illustrate is shown in figure 2. The rule for `isAllowedCredit` contains a condition on the variable `?owner`, suppose no rule exists for `owner=` then SOUL will attempt to interpret that condition as a message. For example in a particular usage of the `isAllowedCredit` predicate, the variable `?account` can be bound to a `BankAccount` object. As `BankAccount` implements an `owner` method, the message `owner` can be sent to the object. The result of the message will be unified with the `?owner` variable. The second condition of the `isAllowedCredit` rule illustrates the use of boolean messages, again supposing there is no rule for the predicate `isOfAge` SOUL will send this as a message to the object in the variable `?owner`. The message should return a boolean value, which is then the success or failure of "proving" the condition.

We have so far concentrated mostly on the interaction *from* SOUL, but true linguistic symbiosis of course goes both ways. In the older SOUL version the Smalltalk-to-SOUL direction of interaction was based simply on the fact that SOUL is an evaluator implemented in Smalltalk. Logic rules could simply be used by creating a `SOULEvaluator` object and sending it an evaluation message with a query to be evaluated passed as a string. This however does not provide for a very clean interaction mechanism and has been replaced in the new SOUL with a mechanism based on translating messages to conditions. This is really the converse of the conditions to messages translation and similarly occurs as an effect of method lookup. When no method is found for a message, the message is translated to an invocation of a predicate instead. Again taking figure 2 as an illustrative example, it is possible to send the message `isAllowedCredit` to a `BankAccount` object even though the `BankAccount` class does not implement a method for it. Upon such a message, Smalltalk will invoke the predicate

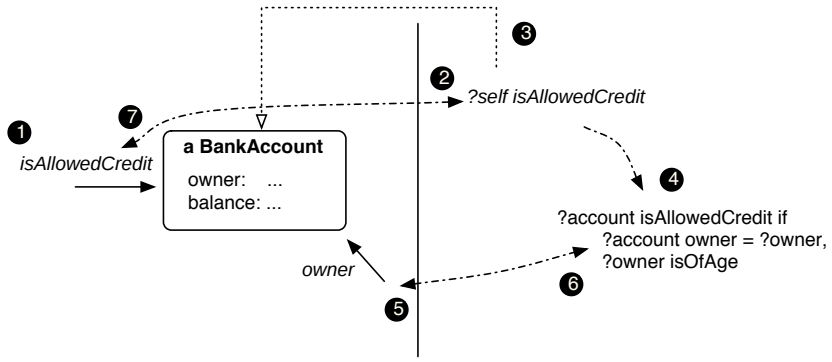

Figure 3: Illustration of an execution trace involving message and condition translation.

`isAllowedCredit` instead, with the first argument of the predicate bound to the receiver of the message. The success or failure of invoking the predicate determines the boolean value of the message.

Figure 3 illustrates the above with an execution trace involving both directions of translations. In a first step (1) the message `isAllowedCredit` is sent to an instance of `BankAccount`, as there is no method to interpret the message with, the message is translated (2) to a query `?self isAllowedCredit` where the (3) variable `?self` is bound to the `BankAccount` object. This query is to be proven (4) using the only rule for `isAllowedCredit`. That rule contains as its first condition one using the predicate `owner=`, because no rule exists for the predicate (5) the condition is translated to a message. The (6) return value of that message then becomes the binding of the variable `?owner`. Proving the second condition of the rule also involves a translation, but it is skipped here to not overcrowd the diagram. Finally when the query of step 2 is finished, (7) true or false is returned as a result of the message of step 1, depending on whether the query was a success or not.

In the case of the `ownedAccount=` predicate the result of invoking it from a message is a bit different. The message `ownedAccount` can be sent to a `BankAccount` instance, which will invoke the `ownedAccount=` predicate with the first argument bound to the instance while the second argument `?account` is left unbound. Thus every solution for the `?account` variable will be determined. These solutions are then returned, in a Smalltalk collection, as the result of the `ownedAccount` message.

Having discussed the more technical aspects of how linguistic transparency is achieved with the new interaction mechanism, we note that the intent for the linguistic symbiosis is to also provide some degree of transparency to the implementation of messages and conditions. It should be possible to implement the handling of a message with a method, but also with logic rules and vice-versa. In particular it should be possible to change to either representation for a particular piece of code without affecting the callers of that code. This transparency is however more difficult due to differences in the two combined paradigms: for example while rules can be used with unbound arguments, this is not possible for methods. It is then usually easier to replace a method with a rule than vice-versa. Some of the issues in linguistic and actual programming

transparency are further discussed in the next sessions.

4 The Case: an Aspect Weaver

In the previous sections, we have first given an outline of how our transaction weaver will be used by the application programmer. Second we have talked about the logic language: SOUL, which together with the LiCoR and Irish libraries, is used in implementing parts of the weaver, and about the logic OO symbiosis. This section is where the rubber meets the road: we will show how logic symbiosis has helped us in implementing a part of the weaver.

Conceptually, we can divide the work of an aspect weaver in two main divisions: first finding the places in the code defined in the crosscut language, and second modifying this code as given by the advice definition. Our aspect weaver only uses logic symbiosis for resolving crosscuts, code modification is performed solely using OO programming. Therefore, we will only describe the crosscut resolving code here.

4.1 Intra-Method Crosscut Definitions

Existing papers on Logic Meta Programming for Aspect-Oriented programming [3, 9] mainly talk about crosscut definitions, and convinced us that it is beneficial to offer the full capabilities of the logic language to the crosscut programmer. This not only allows very powerful crosscuts to be specified, but also offers a straightforward way to abstract these complicated crosscuts into a simple specification.

Although the application programmer will not need to implement these crosscuts himself, this feature was a great help to us in implementing the weaver. Indeed, while the application programmer only needs to specify method signatures, as we have said in section 2, the weaver will need more than this. This is because code will have to be inserted, not only at the beginning and the end of the method body, but also at specific sub-method entities, i.e. within the method body itself.

Luckily, we already have a way in which we can find sub-method entities inside methods, and that is the parse tree traversal introduced in section 3. We can use this to, for example, locate all message sends within a given method, by using the `isMessageSend` rule introduced in section 3. This rule then acts as some kind of filter for the code, allowing only message sends to pass through, when searching the source code using the following rule:

```
weaveSubMethodMatch(?meth, ?filter, ?submethodlist) if
   findall(?submeth,
      traverseMethodParseTree(?meth, ?submeth, ?filter, equals),
      ?submethduplist),
   filterDups(?submethodduplist, ?submethodlist)
```

This rule finds a list `?submethodlist` of all sub-method entities `?submeth`, in the method `?meth`, that satisfy the filter rule `?filter`, which is a part of the crosscut definition, given by the crosscut programmer. In our example, this is the `isMessageSend` rule. These entities are copied into `?submeth` by

the `equals`[3] rule and accumulated in the list `?submethduplist`, which might however include duplicates, due to duplicate actions performed by the inference engine. These duplicates are filtered out by the `filterDups` rule and will result, in our example, in a list of all message sends in the body of the method.

We now have the ability to search for parts of methods, by giving filter rules. As these filter rules are logic rules, we have the full power of the reasoning engine at our disposal, which allows us to write powerful crosscut definition rules. One example is the `sendReceiverIsEJB` filter, which lets all message sends of which the receivers are Enterprise JavaBeans pass through. The code for `sendReceiverIsEJB` is below:

```
sendReceiverIsEJB(send(?,?,?,?rec@(?,?rectypename,0,?),?,?)) if
   interfaceWithName(?rectype,?rectypename),
   interfaceWithName(?ejbremote,javax.ejb.EJBObject),
   superinterfaceOf(?ejbremote,?rectype)
```

Briefly put, this rule matches all message sends of which the receiver type `?rectype`, with name `?rectypename`, is a sub-interface of the interface `javax.ejb.EJBObject`, which by definition are Enterprise JavaBeans. We can now use this filter to obtain a list of all message sends to Enterprise JavaBeans, within a given method, as follows:

```
? EJBMessagesIn: ?method = ?submethodlist if
   weaveSubMethodMatch(?method, sendReceiverIsEJB, ?submethodlist)
```

As it happens, for our transaction aspect, we need to obtain these message sends, because they must be modified. Instead of simply calling the original receiver, these calls must be wrapped with transaction-specific code. While weaving transactional methods, the weaver will first use the above logic rule to obtain these message sends, and second, traverse this list to add the wrapper code.

In general, applying crosscut location rules yields a list of sub-method entities. At these join points code must be woven into the program, and this will be taken care of by the code modification part of the weaver. As we have said above, we do not discuss code modification here, as it is outside of the scope of this paper. Therefore we move on, and discuss an interesting feature of the weaver, made possible by the logic symbiosis: method detection.

4.2 Method Detection

A second use of the parse tree traversal is detection of methods which should probably be made transactional. This feature of the weaver uses properties of the code within a method to determine that this method should be considered by the programmer when writing a crosscut definition.

More concretely: we can investigate the code to find out which methods perform operations on the database, and therefore should be made transactional. We have an implementation of this detection algorithm: the `txMethodsIn:` rule, which uses the parse tree traversal and the `sendReceiverIsEJB` filter as follows:

[3] The code of which is `equals(?x,?x)`

```
?txMethodsIn: ?class  = ?method if
   methodInClass(?method,?class),
   or(fieldRead(?method,?),fieldWritten(?method,?)).

fieldWritten(?method,fieldWrite(?type,?field)) if
   traverseJavaMethodParseTree(?method,?message,
         sendReceiverIsEJB,equals),
   isSetterMessage(?message,?type,?field).

fieldRead(?method,fieldRead(?type,?field)) if
   traverseJavaMethodParseTree(?method,?message,
         sendReceiverIsEJB,equals),
   isGetterMessage(?message,?type,?field).
```

For brevity, we have omitted the definition of the `isGetterMessage` and `isSetterMessage` rules. It suffices to say that these rules verify, by using the EJB naming conventions, that the message `?message` is indeed a getter or setter for the instance variable `?field` of the receiver `?type` of the message. The `txMethodsIn:` rule simply detects all methods of a given class for which such variable access occurs in the method body. If so, the method should be considered for inclusion within a transaction, and therefore it is returned.

This feature is made available to the aspect programmer through the user interface of the weaver, which includes a specific 'Suggestions' button. When clicked, the corresponding Smalltalk method: `detectSuggestions`, on the user interface object is invoked. `detectSuggestions` obtains a collection of classes, specified by the user in the `classesWidget` list widget, and launches the query on that collection of classes. The result collection is then used to populate the `suggestionsWidget` list widget. We show the relevant sections of the methods' code below:

```
detectSuggestions
  |methods |

  aClassCollection := classesWidget list.
  methods := self txMethodsIn: aClassCollection.
  (methods = false) ifTrue: [methods := #()].
  suggestionsWidget list: methods.
```

The code is quite straightforward, save for the `(methods = false)` test. This test is required, because if no methods are detected, the rule will return false, while the suggestions widget expects a collection as argument. Therefore, in such a case, we change `methods` to an empty collection.

The programmer will browse through the list of suggestions displayed in the `suggestionsWidget` list widget, optionally looking at the source code of the method, and will determine which of the suggested methods should indeed be made transactional. Once these transactional methods have been identified, their signatures will be written down in the aspect program, as we have shown in section 2.

From such an aspect program, the weaver will have to determine the transactional attribute specification for each method of the specified class. It turns

out that logic programming is well-suited to tackle this problem, and how this is performed is discussed in the following section.

4.3 Method Crosscut Definitions

In section 2, we have shown how aspect programs are written in our aspect language. Recall that for each class crosscuts are given as a list of method signatures. At weave time, the weaver needs to determine the exact crosscut specification which is applicable for all methods. Sadly, this is not really straightforward, given the interactions between exact method specifications, wildcard specifications and the default class specification.

Our first implementation of the matching mechanism was written in Smalltalk and consists of eleven methods, each with a length of approximately five lines of code. We only realized afterward that we were, in fact, manually performing the kind of work a logic interpreter would do. Therefore, after the logic OO symbiosis had become available, we re-implemented the matching in logic. This turned out to be quite successful: we only have six logic rules, each of about four lines of code. Furthermore, the logic code was much easier to write, we find it more readable than the Smalltalk code, and it only took us one-third of the time to develop.

Below we give the logic rules for the matching mechanism. For each method in the class for which `?classpec` is an aspect program, a transactional method specification will be returned.

```
? selectedMethodIn: ?classpec  = ?methodspec if
   ?classpec exactMatch: ?methodspec .

? selectedMethodIn: ?classpec  = ?methodspec if
   ?classpec wildcardMatch: ?methodspec ,
   ?methodspec themethod = ?method,
   not(?classpec exactMatch: ?spec,?spec themethod = ?method).

? selectedMethodIn: ?classpec  = ?methodspec if
   ?classpec defaultMatch: ?methodspec ,
   ?methodspec themethod = ?method,
   not(?classpec wildcardMatch: ?wspec,?wspec themethod = ?method),
   not(?classpec exactMatch: ?spec, ?spec themethod = ?method).
```

The above logic code determines how a specification is obtained, either through an exact method specification, through a wildcard specification, or through the default class transactional behavior. For wildcard and default matches, the code ensures that, for a method, a general specification (e.g., a wildcard) does not override a more specific one (e.g. an exact specification).

Below is the code which determines if a match occurs:

```
?classpec exactMatch: ?methodspec  if
   ?classpec wovenmethods list = ?speccoll,
   ?speccoll contains: ?methodspec.

?classpec wildcardMatch: ?methodspec  if
   ?classpec wovenwildcards list = ?speccoll,
```

```
   ?speccoll contains: ?methodspec,
   ?spec theclass = ?class, ?spec methodname = ?name,
   methodInClass(?method,?class),
   methodWithName(?method,?name),
   MethodTXSpecs onMethod: ?method withWildcardSpec: ?spec
      = ?methodspec.

?classspec defaultMatch: ?methodspec  if
   ?classspec wovendefaults list = ?speccoll,
   ?speccoll contains: ?methodspec,
   ?spec theclass = ?class,
   methodInClass(?method,?class),
   MethodTXSpecs onMethod: ?method withClassSpec: ?spec
      = ?methodspec
```

From each rule, a `MethodTXSpecs` object is returned, which specifies the transactional specifications for that method. This object is either taken directly from the exact method specification in the aspect language, or cloned from a wildcard or default specification. Note that this code heavily mixes Smalltalk method invocations with logic code. Nevertheless, the code still proved easy to write and to understand, once the programmer is used to the symbiosis mechanism.

4.4 Conclusion

In this section we have shown how the logic and OO symbiosis in Smalltalk and SOUL helped us to implement the crosscut resolving part of our transaction management aspect weaver. We have focused on three parts of the weaver where the symbiosis plays a significant role: firstly intra-method crosscuts, secondly method detection, and thirdly method crosscuts.

Intra-method crosscuts heavily depend on the logic parse tree traversal, as a crosscut definition is given as a filter for the traversal. This allows us to declaratively write down the features of the entities which we are looking for. This makes it straightforward to write down quite advanced filters, as we have shown in our `sendReceiverIsEJB` example. This has proven to be a real asset, provided by SOUL, for implementing the weaver.

Method detection is a feature which is not strictly a required part of a weaver, however it turns out to be useful for the programmer. With method detection, the weaver provides a list of methods deemed interesting, i.e. which might need to be made transactional. We have shown a logic detection algorithm which returns such methods, and how this code is integrated within the user interface. Again, the code is quite straightforward, thanks to the logic OO symbiosis.

For method crosscuts we have really shown the strength of logic symbiosis. We have shown how an existing algorithm, which matches transactional specifications to methods, was re-implemented using logic OO symbiosis. The new code turns out to be shorter, was easer to write and only took one third of the time of the original code to develop. The code shows how, in logic code, Smalltalk and logic code can be combined, leading to true language symbiosis.

5 Discussion

In performing the aspect weaver case study, we came across a number of software engineering issues related to the use of language symbiosis. The SOUL linguistic symbiosis was originally developed in the context of new software involving business rules, which don't make use of LiCoR [5]. The aspect weaver case, however, differs from the above in that an implementation of it already existed using non-symbiotic SOUL, and that it does makes use of the LiCoR library, which also currently does not exploit symbiosis. This raised the issue of how to adapt the implementation in the old syntax, and in particular the existing logic rules, to the new version of SOUL. To tackle this, a simple tool was developed to "rename" and change the order of arguments of predicates in the old syntax to the new one. This tool not only changes the rules implementing the predicate, but also corrects the references to the predicate from within all rules using the predicate.

The "rename" tool, however, currently does not change Smalltalk code using the old method of Smalltalk-to-SOUL interaction to the new symbiotic interaction. This is because the change is not necessarily straightforward, due to two constraints the symbiotic interaction mechanism has in comparison with the old interaction method. First, calling the SOULEvaluator with a query as a string allows any query to be passed, while translating messages to queries means a query can only consist of one predicate. Secondly, in translating messages to predicate invocations only one variable can be considered to be left unbound, which is always the one after the equal sign in the predicate name, while the query-as-a-string method can leave any variable unbound.

Technically, the tool can easily deal with the first constraint simply by making a new rule containing the conditions of the query. Also, from a software engineering standpoint this would be a good solution as the new rule introduces an abstraction for the query. The second issue, however, is technically harder to deal with, especially if linguistic transparency is to be conserved . We should note at this point that the linguistic symbiosis *does* allow for multiple unbound variables in queries that are translations from Smalltalk messages, but in a way that breaks linguistic transparency so we won't discuss it further here, details can be found in [8].

Barring this solution, adapting the code seems best left to the programmer, but is less attractive from a software engineering standpoint. However, in the aspect weaver case this turned out not to be an issue, as we only came across Smalltalk-SOUL interaction code involving queries where only a single variable was left unbound. Therefore, we did not deal with this further.

The case study also pointed out a language design issue in the way queries without solutions are handled. The method `detectSuggestions` shown in section 4.2 sends the message `txMethodsIn:`, the result of which is checked for being `false`, indicating there were no results at all, otherwise the result is a collection of solutions. The code would be simpler if in the case of no solutions the result of the message would simply be an empty collection. The reason this is not the case has to do with a somewhat technical issue of how the symbiosis is currently implemented. At the point where the results from SOUL are translated back to Smalltalk, the distinction between a message that expects a boolean result and one that expects a collection of results is difficult to make in the case of a query that failed. It was therefore previously opted to always

return `false` in that case. This situation can be easily remedied and will be redesigned in a future version of the symbiosis.

We find that writing crosscut definitions, for internal use in the weaver, using logic rules that act as filters on a parse tree to be extremely powerful. We feel that it will be harder to find a more powerful kind of crosscut definition that still remains useable. It is interesting to note that we discovered during the ACP4IS [2] workshop at the AOSD04 conference, that other workshop participants have independently come to a similar conclusion. A result of the discussion group on aspect languages was the idea of using a logic language as a crosscut language: the programmer writes logic rules that select entities of a tree, be it a program tree or a call stack. We therefore think that this would be an interesting avenue for logic OO multi-paradigm programming.

We were impressed with the ease in which we could re-implement the method crosscut code in subsection 4.3. We consider the language symbiosis to be successful, even given its current limitations, based on the results we obtained in subsection 4.3. There was only one important issue that needs to be raised here: the programmer needs to be aware of the difference in evaluation eagerness between Smalltalk and logic code. In Smalltalk, parameters of a message send are evaluated eagerly, while in logic code this is not the case. Therefore, when writing logic rules, but using Smalltalk syntax, initially the programmer might be tempted to chain message sends, which in fact need to be performed in different steps. As an example, consider the `exactMatch:` rule in subsection 4.3. We might be tempted to condense the body into `?classpec wovenmethods list contains: methodspec`, emulating standard Smalltalk notation. However, this does not work, since we are writing logic code, and `?classpec wovenmethods list` will not be evaluated before the logic rule `contains:` is evaluated. This difference in evaluation eagerness can be confusing, but we have quickly learned to take it into account. The key is not to confuse the syntax with the semantics: although we are writing code in a Smalltalk-like syntax, we must keep in mind we are writing logic code and therefore need to remember the properties of the logic paradigm.

6 Conclusion

In this paper we have presented our experience with implementing an aspect weaver for transaction management using SOUL and its newly developed linguistic symbiosis mechanism.

We have first given a general overview of transaction management considered as an aspect. In this overview we summarized why we can consider transaction management as an aspect, and given a number of existing implementations of aspects for this concern. We then outlined the goal of our aspect weaver: providing transaction management for Enterprise JavaBeans, and have shown what our aspect language looks like.

Secondly, we talked about Logic Meta Programming and linguistic symbiosis. We introduced the logic reasoning engine, SOUL, that we employ and have shown how SOUL uses the logic library, LiCoR, to reason about Smalltalk code. Also, we discussed Irish, which allows SOUL to reason about Java code, in some depth. Last in this section, we have detailed the linguistic symbiosis feature of SOUL whereby logic and Smalltalk code can be interleaved.

We have presented the aspect weaver case in more detail. We started with showing how we can write down crosscut specifications for sub-method entities. We have shown how we can easily write down powerful specification using logic rules. Next we have shown how we can investigate the bodies of methods, using logic rules to look for relevant information, and straightforwardly present found methods to the application programmer, thanks to the symbiosis. Last, we presented a re-implementation of the algorithm which matches transactional properties to methods, based on the aspect program. This algorithm is significantly shorter than the previous one, and took only one-third of the time to implement, thanks to the logic OO symbiosis.

Lastly, we have discussed a few software engineering and language design issues encountered in the aspect weaver study as well as some of the advantages and disadvantages of the symbiosis approach. First of all we have the issue of how to adapt existing code to make use of the new interaction mechanism. Second, a small language design issue was uncovered in how queries without solutions are translated back to Smalltalk. However, despite the need to change existing Smalltalk-SOUL interactions in existing code, we had good experience in the case study with what language symbiosis was designed to do: introduce new interactions by refactoring methods to rules and vice versa, with minimal impact on the code using those methods and rules.

To conclude, we have shown how multi-paradigm programming, more specifically logic OO symbiosis has significantly helped us in building an aspect weaver. We have seen that we gained substantial advantages thanks to, on the one hand, the ability to use logic programming to specify crosscuts, and, on the other hand, the logic OO symbiosis, which allows us to intermix logic and Smalltalk code.

Acknowledgments

We thank Andy Kellens for proof-reading and Theo D'Hondt for supporting this research.

References

[1] The AspectJ project. http://eclipse.org/aspectj.

[2] The third AOSD workshop on aspects, components, and patterns for infrastructure software (ACP4IS). http://www.cs.uvic.ca/ ycoady/acp4is04/.

[3] J. Brichau, K. Mens, and K. De Volder. Building composable aspect-specific languages. In *Proc. Int'l Conf. Generative Programming and Component Engineering*, pages 110–127. Springer Verlag, 2002.

[4] K. De Volder. Aspect-oriented logic meta programming. In P. Cointe, editor, *Meta-Level Architectures and Reflection, Second International Conference, Reflection'99*, volume 1616 of *Lecture Notes in Computer Science*, pages 250–272. Springer Verlag, 1999.

[5] M. D'Hondt and K. Gybels. Seamless integration of rule-based knowledge and object-oriented functionality with linguistic symbiosis. In *Proceedings of the 19th Annual ACM Symposium on Applied Computing (SAC 2004),*

Special Track on Object-Oriented Programming, Languages and Systems. ACM Press, 2004.

[6] J. Fabry and T. Mens. Language-independent detection of object-oriented design patterns. *Elsevier Computer Languages*, To Be Published.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: elements of reusable Object-Oriented software*. Addison-Wesley, 1995.

[8] K. Gybels. Soul and smalltalk - just married: Evolution of the interaction between a logic and an object-oriented language towards symbiosis. In *Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Languages*, 2003.

[9] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *2nd International Conference on Aspect-Oriented Software Development*, 2003.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European conference on Object-Oriented Programming*. Springer-Verlag, jun 1997.

[11] J. Kienzle and R. Guerraoui. Aop: Does it make sense? - the case of concurrency and failures. In *Proceedings of ECOOP 2002*. Springer Verlag, 2002.

[12] A. Rashid and R. Chitchyan. Persistence as an aspect. In *2nd International Conference on Aspect-Oriented Software Development*. ACM, 2003.

[13] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of OOPSLA 02*. ACM, November 2002.

[14] R. Wuyts. *A Logic Meta-Programming Approach to Support Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium, January 2001.