Seamless Integration of Rule-Based Knowledge and Object-Oriented Functionality with Linguistic Symbiosis

Maja D'Hondt[*]
System and Software Engineering Lab
Vrije Universiteit Brussel
mjdhondt@vub.ac.be

Kris Gybels[†]
Programming Technology Lab
Vrije Universiteit Brussel
kgybels@vub.ac.be

Viviane Jonckers
System and Software Engineering Lab
Vrije Universiteit Brussel
vejoncke@vub.ac.be

ABSTRACT

Software applications often contain implicit knowledge in addition to functionality which is inherently object-oriented. Many approaches and systems exist that focus on separating *rule-based knowledge* from object-oriented functionality and representing it explicitly in a logic reasoning system. Support for seamless integration of this knowledge with the object-oriented functionality improves software development and evolution. Our hypothesis is that a *linguistic symbiosis* is required between the logic reasoning and object-oriented programming paradigms in order to achieve seamless integration.

This paper presents a symbiotic extension of SOUL, a system which implements a logic programming language and a production system in Smalltalk. The presence of these two logic reasoning systems in SOUL ensures a comprehensive coverage of rule-based reasoning styles, more specifically forward and backward chaining. Our approach is evaluated by means of two case studies. We summarise a comprehensive survey, which shows that existing systems do not fully support seamless integration.

Keywords

Rule-based systems, Multi-paradigm programming, Linguistic symbiosis

1. INTRODUCTION

Software applications often consist of implicit knowledge. Examples are policies, preferences, decisions, processes, workflows and so on. In this paper we focus on rule-based knowledge where each rule is an atomic unit of knowledge that state how to infer knowledge or initiate actions when certain conditions are satisfied [13].

When making rule-based knowledge explicit, the most suitable representation for rules is logical implications in a logic reasoning system [29]. A crucial advantage of such systems is that the flow of rules is automatically managed by a rule engine. The software's core application functionality, on the other hand, is best expressed in an object-oriented programming language. Hence, in this paper we are interested in systems that support both paradigms – logic reasoning and object-oriented programming.

Although our goal is to keep the two paradigms as distinct as possible in order to enhance modularity of rule-based knowledge and object-oriented functionality, the two have to be integrated at certain points in order to obtain one software application. We identify four issues in integrating the logic reasoning system and the object-oriented programming language. However, the level of automation and transparency of the integration can seriously affect the development process: as the application evolves, an object-oriented implementation may very well need to be partially replaced by a logic reasoning representation, and vice versa [31].

We conjecture that in order to support such evolution of representation, a *linguistic symbiosis* [21] [30] [24] is required to integrate the logic reasoning system and the object-oriented programming language. This allows programs to call other programs in a transparent and automatic way, irrespective of the language they are implemented in. As a result, when a program, originally implemented in one language, is instead implemented in the other, the programs that use it do not need to be adapted at all. However, a survey of existing systems shows that this seamless integration is not fully supported by existing approaches [17].

While we have been able to reuse solutions provided by previous research on linguistic symbiosis, we have to resolve new problems caused by the paradigmatic distance between a logic reasoning system and an object-oriented programming language. We introduce linguistic symbiosis between Smalltalk and SOUL [22], which provides a logic programming language and a production system. We evaluate our approach by means of two case studies. Most importantly, this shows that our approach enables incremental development and evolution.

The contribution of this paper is a model which covers

[*]Maja D'Hondt is a research assistant of the *Flemish Institute for the Promotion of Scientific-Technological Research in the Industry*, Flanders, Belgium (I.W.T).

[†]Kris Gybels is a research assistant of the *Fund for Scientific Research*, Flanders, Belgium (F.W.O.).

all the identified issues of seamless integration of a pure object-oriented programming language and a logic reasoning system. We cover both forward and backward reasoning. Furthermore, we provide an implementation of this model in Smalltalk and SOUL.

First of all, this paper illustrates the challenges in integrating rules and object-oriented functionality (Sec. 2). Next, we discuss SOUL and our symbiotic extension (Sec. 3). We briefly present the two case studies used for evaluating our approach and discuss the results (Sec. 4). The section on related work consists of a summary of a comprehensive survey (Sec. 5). Finally, we present our conclusions and future work (Sec. 7).

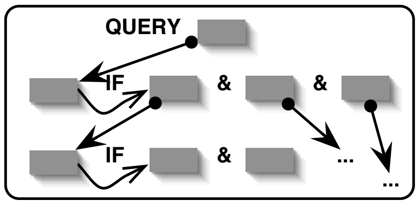
2. INTEGRATING RULES AND OBJECTS

This section first provides some background information on logic reasoning systems. Then, we explain and illustrate the advantages of having a seamless integration between logic reasoning systems and object-oriented programming languages. Next we present and illustrate the four integrating issues we identified. Finally, we introduce the concept of linguistic symbiosis.

2.1 Logic Reasoning Systems

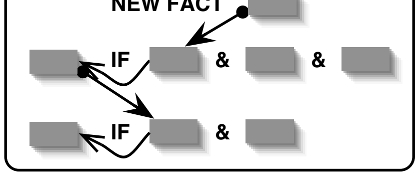
Rule-based knowledge is most suitably represented in a *logic reasoning system*. Two important classes are *logic programming languages* and *production systems* [29]. Both use implications for representing rules explicitly. Implications are logic sentences of the form *antecedent* ⇒ *consequent*, also known as *conclusion IF premise 1* & ... & *premise n*.

In logic programming languages, *backward chaining* is used to prove a conclusion by attempting to establish the premises. The figure below illustrates traditional backward chaining. A premise is established if a rule is found that can conclude



it. Traditional backward chaining is triggered by performing a query which either results in a successful proof and a set of bindings for any unbound variables in the query, or fails.

In traditional production systems, *forward chaining* is triggered if a new fact is asserted and establishes a rule's premise. The figure below illustrates traditional forward chaining. If all of a rule's premises are established in this



way, the rule's conclusion is generated and asserted as a new fact. Again, this can result in other rules being triggered.

Later on in this paper, the above figures are extended to show how traditional chaining and object-oriented message passing are adapted to enable linguistic symbiosis.

2.2 Development with Seamless Integration

There exist guidelines and methodologies for developing object-oriented software applications with explicit (business) rules [31] [28] [16]. They stress that during application development or evolution, an object-oriented implementation may very well need to be partially replaced by a logic reasoning representation, and vice versa. Because such iterative or incremental development is even more heavily promoted by eXtreme Programming, this becomes an increasingly important issue.

Consider a simplified e-commerce application which supports online business-to-customer sales. The Smalltalk implementation of the example method `price` on instances of `Order` below. This method obtains the total price of the order and the discount to which the customer is entitled. It subtracts the discount percentage from the total price and returns the result.

```
price
  ^self totalPrice*(100-customer discount)/100
```

Conceptually, a number of rules describe how to derive a customer's discount. Initially, these can be implemented in Smalltalk as a method on the class `Customer` which returns a discount percentage based on properties of the customer or other objects. In the example below, a discount of 5 percent is returned if the customer is loyal.

```
discount
  self isLoyal ifTrue:[^5].
  ...
```

As the rules evolve and increase in number, managing the control flow manually becomes cumbersome. Therefore, a logic reasoning system is employed for representing the rules explicitly. The representation of such a rule in SOUL is shown below. Note that SOUL will be presented in more detail in Sec. 3.

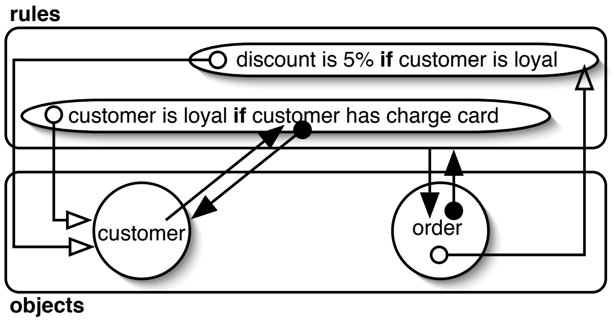
```
?c discount=5 if ?c isLoyal
```

A seamless integration of the object-oriented programming language and the logic reasoning system allows programs to call other programs in a *transparent* and *automatic* way, irrespective of the language they are implemented in. As a result, when a program, originally implemented in one language, is instead implemented in the other, the programs that use it do not need to be adapted at all. Hence, the implementation of the first method `price` does not depend on the actual representation of the discount rules. The invocation of `discount` will be delegated to the object-oriented language if it is implemented as a method and to the logic reasoning system if it is represented as rules.

Note that although this simple example illustrates seamless integration, the single example rule is not meant to demonstrate the need for logic reasoning.

2.3 Integration Issues

The four issues in integrating logic reasoning and object-oriented programming are illustrated below.


rules
discount is 5% if customer is loyal
customer is loyal if customer has charge card
customer
order
objects

1. **How are rules triggered?** For example, an object *order* calculates the price of the purchased items of another object *customer* and discounts the price with a certain percentage. The discount is inferred by the rules, which somehow have to be triggered. This is depicted by the black arrow starting from *order*.

2. **How do rules refer to objects from the core application?** In the example, the rules refer to the object *customer*. This is denoted by the white arrows.

3. **How do rules invoke methods on objects in the core application?** The example shows that the rules need information about the object *customer* in order to infer the discount. The black arrow denotes the invocation of the method that establishes whether *customer* has a charge card or not. Somehow the result has to be returned to the rules.

4. **How are objects informed of the inference results of the rules?** The object *order* needs to refer to the inferred *discount* of *customer*, depicted by the white arrow.

2.4 Linguistic Symbiosis

A *linguistic symbiosis* between two languages enables programs implemented in one language to call programs implemented in another language *transparently* and *automatically*. Transparency is achieved by hiding the mechanisms used by the languages for invoking each other. Automation is achieved by conceiving a one-to-one mapping between the mechanisms of both languages for invoking their own behaviour. As such, linguistic symbiosis allows replacement of parts of a program with another program implemented in a different language, possibly of a different paradigm. This replacement does not require the first program to be adapted at all. Systems that provide a linguistic symbiosis between two languages usually implement one language in the other.

The term first appears in the work on a meta-object-protocol-enabled interpreter written in C++, which can have all of its parts replaced with parts written in the actual language it interprets [21]. A similar scheme is presented in the work on reflective extensions of open programming languages [30, 24]. In these approaches both languages are object-oriented.

While we have been able to reuse solutions provided by the aforementioned research on linguistic symbiosis, we are also presented with issues which arise when integrating logic reasoning with object-oriented programming. In the next section we introduce our symbiotic extension of SOUL and show how we deal with the four integration issues.

3. LINGUISTIC SYMBIOSIS IN SOUL

In this section we first introduce the logic reasoning systems which comprise SOUL. Then we explain how we addressed the four integration issues in order to achieve linguistic symbiosis in SOUL: triggering rules and invoking methods are discussed together, followed by an explanation of exchanging inference results and objects.

3.1 SOUL

SOUL implements logic reasoning in Smalltalk, consisting of both a logic programming language and a production system.

Over the years, a number of people have been involved in the development and use of the Prolog-like logic programming language of SOUL. It is used as a research platform for applying logic programming to a number of software engineering problems: explicitly representing domain knowledge in object-oriented applications [18]; reasoning about the design of object-oriented applications [33, 32]; checking, enforcing and searching for occurrences of programming patterns [22]; supporting evolution of software applications [23]; and architectural conformance checking [34]. Note that although Prolog differs from standard backward-chaining rule-based languages, they share the features that are relevant for integrating backward-chaining logic reasoning with object-oriented programming.

We developed the production system of SOUL to complement the goal-oriented style of logic programming with a data-oriented and generative style of logic reasoning. It is inspired by the new generation of production systems such as OPSJ [8], JRules[9] and CafeRete[1], which in turn are object-based versions of the older production systems OPS5 [12] and CLIPS [3].

The representation of rules is essentially the same for both of SOUL's logic reasoning systems, with the exception of some specialised constructs inherent to Prolog-like logic programming, such as the *cut*. A characteristic that SOUL has in common with most systems that combine logic reasoning and object-oriented programming, is the ability to reason about objects instead of literals such as strings and numbers.

Neither the logic programming language nor the production system of SOUL present contributions in se, since they are based on existing, successful languages and systems. Their combined presence in SOUL ensures a comprehensive coverage of rule-based reasoning styles. Our contribution is the integration of these rule-based reasoning styles with object-oriented programming.

3.2 Triggering Rules and Invoking Methods

Our goal is fully automatic and transparent dispatching between SOUL and Smallltalk. An object-oriented program can dispatch a subtask to another object-oriented program, or the exact same program can dispatch it to the logic reasoning system by triggering rules. Similarly, when the logic reasoning system infers a goal or facts, it can either trigger other rules, or the exact same rule can invoke a method.

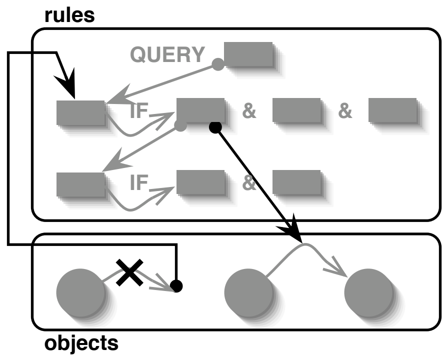
Since backward and forward chaining have mostly different dispatching strategies, we describe the two approaches separately.

3.2.1 *Backward Chaining*

We obtain automatic dispatching by devising a one-to-one correspondence between logic predicate names and method names. In order to achieve this, we adapt the syntax of predicates to resemble that of Smalltalk's keyword-based method invocation syntax. Hence, we write:

```
?c canBuy: ?p if ?c inEurope & ?p intlShipping
```

To obtain a transparent mapping between rules and methods in SOUL, the mechanisms used for invoking each other are hidden. We do this by adapting standard method dispatching and rule chaining respectively. This is shown in the figure below.

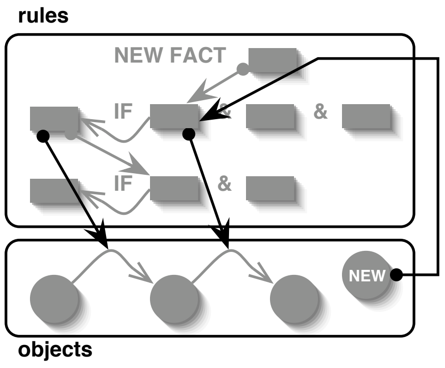


In SOUL, backward chaining is triggered from Smalltalk when an undefined method is invoked. As a result, SOUL performs a query whose predicate name equals the method name. The receiver is bound to the query's first variable and the actual parameters are bound to the other query variables. For example, the message `canBuy:` is sent to an instance of `Customer` with an instance of `Product` as parameter, but Customer objects do not implement this method. As a result, the query `?c canBuy: ?p` is performed with the instance of `Customer` bound to `?c` and the instance of `Product` bound to `?p`.

In SOUL, the backward chaining rule engine first attempts to establish a rule's premise by finding a rule which could prove the premise. When such a rule is not defined, SOUL invokes a message whose name equals the predicate name in the premise. The message is sent to the object bound to the premise's first variable. The actual parameters of the message are the objects bound to the corresponding variables of the premise. For example, when trying to prove the first premise of the aforementioned rule which concludes `canBuy:`, but no rule is defined which can establish the premise `?c inEurope`, the message `inEurope` is sent to the object bound to `?c`.

3.2.2 *Forward Chaining*

For automatically invoking Smalltalk methods from SOUL, the forward chainer uses the same mechanism as the backward chainer: the one-to-one mapping from predicate names to method names. However, an alternative mechanism is used for triggering forward-chaining rules from Smalltalk. Since objects are considered to be facts, their state changes are monitored. SOUL interprets a change as a new fact being asserted, which triggers the forward chainer automatically and transparently. SOUL is able to restrict monitoring to specific attributes of specific objects, which reduces the number of times the forward chainer is triggered. Consider the following example: SOUL monitors the attribute `chargeCard` of instances of `Customer`. When an object's charge card is set, it triggers the forward chainer. The figure below illustrates (amongst others) how Smalltalk triggers SOUL.



The figure also illustrates how SOUL invokes methods. SOUL's forward chainer attempts to establish a premise after a changed object is bound to one of its variables. This is achieved by trying to invoke a message, which is similar to the backward chainer's approach. If this fails, the traditional forward-chaining approach is taken. Once all the premises of a rule have been established, the forward chainer tries to conclude the rule by invoking a method. The method name equals the predicate name in the conclusion and the objects bound to the conclusion's variables are again used as actual parameters. If the method is not defined in the receiver's class, the bound conclusion is asserted as a new fact. To illustrate this, consider the rule below.

```
?c isLoyal if ?c hasChargeCard
```

Since, the forward chainer is triggered because an instance of `Customer` has changed, the object is bound to the variable `?c`. The engine tries to establish the premise by invoking the method `hasChargeCard` on the object. If this succeeds and the rule fires [1], SOUL draws the conclusion by attempting to invoke the method `isLoyal` on the `Customer` object bound to `?c`. If this fails, `?c` is substituted by the `Customer` object in the conclusion, and this is asserted as a new fact.

3.3 Exchanging Objects and Inference Results

Returning values from Smalltalk as a result of invoking a method from SOUL occurs both in backward and forward chaining. Exchanging values in the other direction is typically only used in backward chaining, since the forward chainer "returns" values implicitly as side-effects of drawing conclusions. Moreover, the way backward chaining is triggered poses an additional challenge. Therefore

[1]Production systems do not fire all rules, but store them in a conflict set and employ some strategy for selecting rules to fire when chaining has stopped. SOUL also takes this approach.

we first discuss returning objects from Smalltalk to SOUL for both forward and backward chaining, then we elaborate on returning inference results from SOUL to Smalltalk for backward chainer alone.

3.3.1 Forward and Backward Chaining

When establishing a premise by invoking a method, either true or false should be returned by Smalltalk, which is interpreted as success or failure by the rule engine.

Non-boolean results from Smalltalk are returned to SOUL using the equality construct `=`. In the example rule below, `totalPurchases` is sent to an object bound to `?c` and the result is unified with `?p`.

```
?c premierCustomer if
  ?c totalPurchases = ?p &
  ?p > 100
```

3.3.2 Backward Chaining

A triggered rule signals either success or failure to the triggering method, which is interpreted as true or false by Smalltalk.

Returning non-boolean values from SOUL to Smalltalk is illustrated by the example rule below. This rule is triggered when the undefined message `discount` is sent to an object and a non-boolean result is expected. When the rule fires because its premise is established, the right hand side of the equality construct in the conclusion is interpreted as the return value of the message send. Hence `10` is returned to Smalltalk.

```
?c discount = 10 if ?c premierCustomer
```

Note that our symbiotic extension of SOUL allows queries with unbound variables to be triggered from Smalltalk so as to use the full power of the SOUL's logic reasoning. In order to do this, SOUL allows messages, which trigger backward chaining, to be sent to or using uninitialised variables. The rule engine binds solutions to those variables during the inference process. For example, it is possible to implement the method `requestMultimediaComputer` as shown below. The temporary variable `c` is not initialised before invoking the method.

```
requestMultimediaComputer
 | c |
 c multimediaComputer.
 ^c
```

This feature leads to a style of programming in Smalltalk that is not natural to the language and thus reveals the logic reasoning paradigm. Nevertheless, our experience is that this mechanism is useful. We leave it up to the programmer to omit these constructs in order to maintain true Smalltalk programming, or to employ them and take advantage of SOUL's full logic reasoning capabilities.

3.4 Implementation

As discussed, linguistic symbiosis in SOUL and Smalltalk is done by transparently mapping methods and rules. Implementing this thus required us to change the behavior of method and rule lookup. Note that we did not have to add language constructs as this would go against the idea of linguistic symbiosis.

For triggering backward chaining the method lookup in Smalltalk is changed, something that is quite easy to do through its reflection capabilities. When the default lookup does not find a method for a message, a `doesNotUnderstand:` message is sent. The method for `doesNotUnderstand:` normally signals an error, but we changed it so that it translates the message to a rule invocation. As SOUL does not support reflection we had to change the rule lookup in SOUL itself. In both cases the lookup happens at run-time and is late bound. For triggering forward-chaining rules from Smalltalk specific attributes of certain objects are monitored. The monitors are installed before the application is run, but can be changed at run-time. We used *AspectS*, a framework for aspect-oriented programming in Smalltalk, to implement the monitors [25].

Late binding, reflection, aspect-oriented programming, or even one language being implemented in the other are by no means necessary for implementing linguistic symbiosis. For example, in combinations of C++ with other languages and in .NET, language combination is done by mapping them all to a single virtual machine which supports language interoperation. One could also take two languages with compilers implemented in C and combine those, although this would present some extra difficulties in allowing values from the one language to be used in the other. The way the combination is implemented is not central to the idea of linguistic symbiosis and it does not add more execution overhead, what is crucial is the idea of making it as transparent as possible which happens mostly on the language design rather than implementation level. Our discussion of our particular prototype implementation is thus brief, we refer to the paper *SOUL and Smalltalk - Just Married* [20] for more details.

4. EVALUATION

First of all, in this section we briefly introduce the two case studies we performed and the criterium used to evaluate our approach. Then we describe the most notable results of the evaluation.

4.1 Case Studies

In what follows, the numbers are approximations and lines of code are always counted without comments and blank lines.

4.1.1 E-Commerce

The first case study consists of developing a software application from scratch, a modest e-commerce application. The functionality we implemented is based on that of existing online business-to-customer stores. However, functionality concerned with persistency, distribution, user interface and such is omitted from this case study. The implemented functionality comprises 300 lines of code organised in 30 classes.

The domain of e-commerce typically contains many business rules [19]. We represented 25 rules for price discounting, lead time to place an order, cancelling orders, creditworthiness, delivery restrictions.

4.1.2 Version Management

The second case study consists of an in-depth analysis of an existing, complex software application, *Store*, a version management system for Smalltalk. This system is widely used with Cincom's VisualWorks, and integrated development environment for Smalltalk. A comprehensive user guide is available, although the code itself is not so well-documented.

Store consists of 250 classes and 70 extensions to existing classes in VisualWorks, which adds up to more than 25000 lines of code.

Store is not developed with explicit rules in mind, but we identified several sources of implicit, rule-based knowledge. For the case study we concentrated on policies for user management and ownership, and on load analysing. The policies concern blessing levels, merging, ownership, packages, prerequisites, publishing and versions. Load analysing is a very generative process which starts with analysing a top-level component and propagates to its (nested) parts. These functionalities are implemented in 350 lines of code organised in 20 classes. From these two areas we distilled close to 70 backward and forward chaining rules respectively, which are triggered in approximately 200 places in the code.

4.1.3 Evaluation Criterium

The main criterium used for evaluating our approach is the level of change propagation as a result of changing the implementation language of a certain sub task of the software application. We do not consider changes for making a certain sub task modular in the implementation. Once a certain sub task is modularised using the procedural abstractions of the paradigm (object and methods in the object-oriented programming language, rules in the logic reasoning system), we attempt to express it in the other language. Obviously, this is only performed in cases where it is desirable to do so. For example, in the second case study, we first identified sources of implicit rule-based knowledge, implemented in Smalltalk, and attempt to express them in SOUL. After this change of implementation language, we note how this change propagates to other places in the code.

4.2 Incremental Development

During application development or evolution, an object-oriented implementation may very well need to be partially replaced by a logic reasoning representation and vice versa, which we refer to as incremental development and evolution (Sec. 2.2).

The evaluation shows that when a certain sub task, originally implemented in one language, is instead implemented in the other, the programs that use it do not need to be adapted at all. In other words, change propagation as a result of incremental development and evolution is non-existent due to linguistic symbiosis.

Previous versions of SOUL, as well as other systems we surveyed, have to cope with the change propagation issue, since client code that activates a certain sub task is different depending of the implementation language of the sub task. Hence, a change of implementation language propagates to the client code.

4.3 Unanticipated Object Attributes

Another form of change propagation occurs when rules evolve and as a result certain object attributes – properties of and relations between objects – have to be added to the object-oriented functionality.

Modern production systems express facts as object attributes and perform actions on these attributes in the conclusion of a rule. Hence, if rules change and object attributes are not anticipated in the object model, this change propagates to the object-oriented functionality. In our approach, this change does not propagate because object attributes that are not reflected in the object model are asserted as facts.

4.4 Extending Object Interfaces

The premises of a rule consist of expressions about objects. These premises can be established by invoking a method. A noticeable issue when developing applications with explicit rules is the need to extend object interfaces in order to provide the appropriate methods. This is an aesthetic change since It is usually possible to obtain the information using the existing object interface and some operators provided by the rule language (such as the equality construct). However, this leads to an object-oriented programming style in the rule language. It is more elegant to extend the object interfaces and provide methods that immediately answer the appropriate question.

Since different rule sets are often used to represent alternative policies or requirements, this practice results in object interface extensions for each rule set. We observe that a mechanism for managing these extensions together with the corresponding rule sets would be useful.

5. RELATED WORK

This section gives a compressed account of a full-fledged survey described in [17].

The surveyed systems all integrate a logic reasoning system with an object-oriented programming language. Most systems are extensions of standard object-oriented programming languages with a forward-chaining production system. The following are commercial systems: *OPSJ* from Production Systems Technology [8], *JRules* from ILOG [9], *Cafe Rete* from Haley [1], *Blaze Advisor* from HNC [5] and *QuickRules* from Yasutech [11]. Non-commercial systems are *CommonRules* [2], *Jess* [10] and *NeOpus* [27]. *KnowledgeWorks* [7] and Aion [6] are representation languages consisting of rule-based and object-oriented programming features. *Prolog++* [26] is an object-oriented extension of Prolog. *Kiev* [4] extends an object-oriented programming language with logic programming.

For each integration issue discussed in Sec. 2.3, we summarise the different approaches. The sign at the beginning of each approach (+, ± or −) indicates its level of transparency and automation. Note that these are generalisations of the mechanisms provided by the surveyed systems, and that subtleties are lost.

1. **triggering rules**

   + Transparent rule triggering is supported (Kiev and Prolog++).

   ± A special function or construct is provided to trigger the rule engine (KnowledgeWorks and Aion).

   − Logic reasoning is implemented as a library in the object-oriented programming language and the library's API is called to set up the rule engine and to trigger the rules (OPSJ, JRules, Cafe Rete, Blaze Advisor, QuickRules, Jess and CommonRules).

2. **objects in rules**

   + A mechanism is provided to generate the objects of a specific class as possible solution values for a logic variable, and message passing is used to manipulate the objects which maintains encapsulation (NeOpus).

± Object state can be manipulated directly which breaks encapsulation (OPSJ, JRules, Cafe Rete, QuickRules, Jess, Aion, KnowledgeWorks, and Blaze Advisor), though this can be avoided in some of these systems as they also support invoking methods from the logic base.

− A specification is required for mapping objects to traditional facts (CommonRules).

3. **invoking methods**

+ The logic reasoning system is extended to allow snippets of object-oriented code in the rules which are evaluated by executing them in the object-oriented language (OPSJ, JRules, Cafe Rete, QuickRules, Aion, KnowledgeWorks, Prolog++ and Kiev).

± Special predicates are provided to invoke object-oriented behaviour (Jess).

− Mappings of predicates to methods of specific classes must be defined (CommonRules).

4. **inference results in objects**

+ Inference results are side-effects of the methods invoked by the rules (OPSJ, JRules, Cafe Rete, QuickRules, Blaze Advisor, Jess, Aion and KnowledgeWorks).

± In addition to the above, a list of changed objects is returned (QuickRules).

− Inference results are returned as logic variables and their bindings (Kiev and Prolog++).

Kiev and Prolog++ do provide transparent and automatic approaches to most of the issues above where other systems do not. However, Prolog++ is basically a logic programming language extended with object-oriented programming features. Therefore, it merely simulates object-oriented programming while maintaining an inherently logic programming style. When developing object-oriented software applications, a full-fledged object-oriented language and development environment are needed.

Of all systems Kiev comes closest to SOUL with respect to the mechanisms for linguistic symbiosis, but Kiev only provides backward chaining. Kiev seems promising, but only limited documentation and use of the system is provided which makes it hard to investigate. A subtle but important difference is our mechanism for mapping the results of the query to the implicit return value of the message send. This is lacking in Kiev, which breaks transparency and thus inhibits incremental development and evolution.

6. DISCUSSION

We can extract from our experience with SOUL and Smalltalk some observations on how to integrate an object-oriented programming language and two important classes of logic reasoning systems seamlessly through linguistic symbiosis. Most importantly, rather than using library calls or adding new language constructs, the syntax of the two languages needs to be made more alike to allow the transparent invocation of methods and rules. The impact of this change is especially evident in the Smalltalk and SOUL case, whereas combining Java and Prolog has less of an impact as their syntaxes are already quite alike. Still, syntactic support for the concept of having a receiver for a predicate or message needs to be added. In some of the systems we surveyed this is done for the logic reasoning language. However, they do not exploit this to a full two-way symbiosis: the language integration is typically more seamless in the direction towards the object-oriented language. As most systems use Java as the object-oriented language, we suspect that Java's lack of reflection is to blame for this imbalance: while reflection is not necessary to make the changes to method lookup, it is more difficult to do without it. Hence many surveyed systems use a library-calling approach to achieve the integration. We chose to use Smalltalk exactly for its support of reflection and because its syntactic constructs focus on the essence of object-oriented programming.

7. CONCLUSIONS AND FUTURE WORK

The differences in integration between the logic reasoning systems is due to the type of rule chaining employed: backward or forward chaining. Since SOUL implements both approaches this paper presents a comprehensive coverage of rule-based reasoning styles and how to integrate them seamlessly with object-oriented programming languages.

SOUL's logic reasoning is based on widely-used languages, more specifically Prolog and state-of-the-art production systems, hence it has the same expressiveness.

Only one feature is introduced in SOUL that breaks transparency: uninitialised temporary variables for representing unbound logic variables in Smalltalk. It is the programmer's choice to omit this feature in order to maintain transparency, or to employ it in order to enable SOUL's full logic reasoning capabilities.

We evaluated our approach by conducting two case studies. We conclude that the seamless integration facilitates incremental development of object-oriented applications with explicit rules. Moreover, we found that our production system with its use of traditional facts in addition to objects provides an elegant solution to unanticipated object attributes. On the other hand, the evaluation also revealed a few opportunities for improvement, such as support for specialised configuration of the default integration and support for managing the object interface extensions required by the ever-evolving rules.

Future work consists of realising the proposed improvements. We observe that *aspect-oriented programming* techniques can contribute here, as is validated by earlier experiments with existing aspect-oriented languages for connecting rules to object-oriented applications [14, 15]. In this work, however, the rules are not expressed in a logic reasoning system. Therefore, we plan to extend SOUL with the necessary aspect-oriented mechanisms for fully supporting unanticipated evolution of rules.

8. ACKNOWLEDGMENTS

We thank Theo D'Hondt for proofreading this paper. We are also grateful to Wim Lybaert, Michel Tilman and Roel Wuyts for their help with the case studies. We thank the anonymous reviewers for their comments.

9. REFERENCES

[1] *Cafe Rete*. Web Site of The Haley Enterprise Inc. at http://www.haley.com.

[2] *CommonRules*. Web Site of IBM Research, http://www.research.ibm.com/rules/commonrules-overview.html.

[3] *CLIPS 6.0*, 1993. User Guide by Joseph C. Giarratano, NASA.

[4] *Kiev 0.9*, 1998. Language Specification by Maxim Kizub, http://forestro.com/kiev/ kiev.html.

[5] *Developing Real-World Java Applications with Blaze Advisor*, 1999. Technical White Paper from HNC Software Inc.

[6] *Aion 9.0 Rules Guide*, 2001. User Guide from Computer Associates.

[7] *LispWorks KnowledgeWorks and Prolog*, 2001. User Guide from Xanalys Inc.

[8] *OPSJ 4.1*, 2001. Manual by Charles L. Forgy from Production Systems Technologies Inc.

[9] *JRules 4.0*, 2002. Technical White Paper from ILOG.

[10] *Jess 6.1, The Rule Engine for the Java Platform*, 2003. User Guide by Ernest J. Friedman-Hill, Sandia National Laboratories.

[11] *QuickRules 2.5*, 2003. Application Developer Manual from YASU Technologies Inc.

[12] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming.* Addison-Wesley, 1985.

[13] Business Rule Group. *Defining Business Rules: What Are They Really?*, 2001. http://www.businessrulesgroup.org/.

[14] M. A. Cibrán, M. D'Hondt, and V. Jonckers. Aspect-oriented programming for connecting business rules. In *Proceedings of the 6th International Conference on Business Information Systems*, 2003.

[15] M. A. Cibrán, M. D'Hondt, D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo for linking business rules to object-oriented software. In *Proceedings of International Conference on Computer Science, Software Engineering, Information Technology, e-Business, and Applications (CSITeA'03)*, 2003.

[16] C. Date. *What not How: The Business Rules Approach to Application Development.* Addison-Wesley, 2000.

[17] M. D'Hondt. A survey of systems that integrate logic reasoning and object-oriented programming. Technical report, Vrije Universiteit Brussel, 2003.

[18] M. D'Hondt, W. D. Meuter, and R. Wuyts. Using reflective logic programming to describe domain knowledge as an aspect. In *First Symposium on Generative and Component-Based Software Engineering*, 1999.

[19] B. N. Grosof, Y. Labrou, and H. Y. Chan. A declarative approach to business rules in contracts: courteous logic programs in XML. In *In Proceedings of the first ACM conference on Electronic commerce*, pages 68–77. ACM Press, 1999.

[20] 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.

[21] Y. Ichisugi, S. Matsuoka, and A. Yonezawa. A reflective object-oriented concurrent language without a run-time kernel. In *Proceedings IMSA: Reflection and Meta-Level Architectures*, 1992.

[22] K. Mens, I. Michiels, and R. Wuyts. Supporting software development through declaratively codified programming patterns. In *Proceedings of the 13th SEKE Conference*, 2001.

[23] T. Mens and T. Tourwe. A declarative evolution framework for object-oriented design patterns. In *Proceedings of Int. Conf. on Software Maintenance*, 2001.

[24] W. D. Meuter. The story of the simplest mop in the world, or, the scheme of object-orientation. *Prototype-Based Programming (eds: James Noble, Antero Taivalsaari, and Ivan Moore)*, 1998.

[25] R. Hirschfeld. Aspects – Aspect-Oriented Programming with Squeak. In *Lecture Notes in Computer Science: Objects, Components, Architectures, Services, and Applications for a Networked World: International Conference NetObjectDays (NODe 2002), Erfurt, Germany*, pages 216 – 232. Springer-Verlag Heidelberg, 2002.

[26] C. Moss. *Prolog++, The Power of Object-Oriented and Logic Programming.* Addison-Wesley, 1994.

[27] F. Pachet and J.-F. Perrot. Report on the néopus system experience.

[28] R. G. Ross. *Principles of the Business Rule Approach.* Addison-Wesley, 2003.

[29] S. Russel and P. Norvig. *Artificial Intelligence, A Modern Approach.* Prentice Hall, 1995.

[30] P. Steyaert. *Open Design of Object Oriented Languages.* PhD thesis, Vrije Universiteit Brussel, 1994.

[31] B. von Halle. *Business Rules Applied.* Wiley, 2001.

[32] R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of TOOLS-USA '98*, 1998.

[33] R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation.* PhD thesis, Vrije Universiteit Brussel, 2001.

[34] R. Wuyts and K. Mens. Declaratively codifying software architectures using virtual software classifications. In *Proceedings of TOOLS Europe'99*, 1999.