CO-EVOLUTION OF OBJECT-ORIENTED SOFTWARE DESIGN AND IMPLEMENTATION

Theo D'Hondt – Kris De Volder – Kim Mens – Roel Wuyts
Programming Technology Lab – Vrije Universiteit Brussel

Abstract: Modern-day software development shows a number of feedback loops between various phases in its life cycle; object-oriented software is particularly prone to this. Whereas descending through the different levels of abstraction is relatively straightforward and well supported by methods and tools, the synthesis of design information from an evolving implementation is far from obvious. This is why in many instances, analysis and design is used to initiate software development while evolution is directly applied to the implementation. Keeping design information synchronised is often reduced to a token activity, the first to be sacrificed in the face of time constraints. In this light, architectural styles are particularly difficult to enforce, since they can, by their very nature, be seen to crosscut an implementation. This contribution reports on a number of experiments to use logic meta-programming (LMP) to augment an implementation with enforceable design concerns, including architectural concerns. LMP is an instance of hybrid language symbiosis, merging a declarative (logic) meta-level language with a standard object-oriented base language. This approach can be used to codify design information as constraints or even as a process for code generation. LMP is an emerging technique, not yet quite out of the lab. However, it has already been shown to be very expressive: it incorporates mechanisms such as pre/post conditions and aspect-oriented programming. We found the promise held by LMP extremely attractive, hence this paper.

1. INTRODUCTION

Recent times have seen a consolidation of methods and tools for software development in production environments. A good number of de facto standard tools have emerged, not the least of which is the *Unified Modelling Language* in some commercial incarnation. At last, one could say, we have come to know the process by which the design and implementation of a complex piece of software is charted. Objects have been widely accepted, the programming language babel seems more and more controlled by the emergence of Java and previously untamed regions of information technology, such as distribution, co-ordination and persistence, are starting to become daily fare in software applications.

So why is software development still arguably the least predictable of industrial processes? Why can comparable software projects, executed by development teams with comparable skills, not be planned with comparable margins of error? Why is our appreciation of the software development process still flawed, even after the introduction of all these new techniques and tools?

For some time now, grounding the development of software in a programming language has proved not to be scalable. This led to the notion of *software architectures* as a collection of techniques to buttress this development process, particularly in those places where programming languages or tools fail to capture the macroscopic structure of the system that

needs to be built. Close to the programming language technology itself, we find the well-understood *framework* approach; at a more abstract level we find techniques built on various kinds of *patterns* and *contracts*.

The latest landslide in this fight for control over software complexity is the emergence of *component* technology. At a time when commercial component toolkits such as *Enterprise Java Beans* are proposed as the solution to our problems, we do well in realising that the advent of components amounts to an acknowledgement of defeat. In fact, by accepting this technique of decomposition into static components, we have come full circle and reinvented data abstraction. The task of making components co-operate is not any better understood than any of the numerous software building strategies we have taken under consideration these past 20 years.

An important step in understanding this partial failure is the insight that software is fluid. It is in constant evolution under the influence of ever changing conditions; software development is sandwiched between a technology that is evolving at breakneck speed, and requirements that must follow the economic vagaries of modern society. In this, the commercial product called *software* is unique; the closest professional activity to that of software developer is that of composer in 18th century Europe. At that time, relatively widespread knowledge of harmony or counterpoint made up for the necessary skills to use and reuse fragments of sophisticated musical artefacts. For instance, [LD96] offers insight on how *invention*, a term borrowed from rhetoric, drives composition according to a process which bears a striking resemblance to building complex computer applications. Unfortunately, equivalent skills needed to master a software artefact are today in far more limited supply than 250 years ago.

This contribution is a synthesis of recent work performed by various people within our lab in addressing this need for more control over the evolution of software. Although in the past, a significant amount of work focussed on the need to *document* evolution and build *conflict detection tools* [SLMD96, CL97], to *recover* architectural information from implementations [KDH98] and to *formalise* the evolution process [TM99], we will concentrate here on an emerging approach for *steering* evolution. This is very recent work and as such has only resulted in experiments and prototypes. We feel however that it is sufficiently mature and promising to be presented here as a whole. In the bibliography we limit ourselves to a number of key documents[1] describing these activities; these in turn contain a much more comprehensive list of references.

We have chosen to use the term *co-evolution*, implying that managing evolution requires the synchronisation between different layers (or views) in the software development process. We will therefore dedicate the next section to an analysis of this statement. Next, we propose a concept called *Logic Meta Programming* (or LMP for short) as a development framework in which to express and enforce this synchronisation process. Another section of this paper will be used to introduce LMP and to situate it in the broader context of software development support. Finally, several experiments with LMP will be presented in evidence of its applicability. We will discuss using LMP as a medium for supporting *aspect oriented programming*, for enforcing *architectural concerns* in an object oriented programming environment and to express constraints on the protocol between a collection of interacting software components. This is by no means a complete coverage of LMP, nor even of the experiments conducted at our lab; we feel however, that it provides sufficient insight in the

[1] Available via http://prog.vub.ac.be

applicability of LMP to the co-evolution of software, while avoiding exposing the reader to too much detail.

2. SYNCHRONIZING DESIGN AND IMPLEMENTATION

Currently accepted procedures in the development of software involve the consideration of several views. In descending order of abstraction one encounters *requirements capture*, *analysis*, *design*, *implementation*, *documentation* and *maintenance.* Not quite by accident, this also happens to constitute an ordering according to increasing level of detail, albeit not a continuous one. There is in fact a kind of watershed between design and implementation, which commits the developer to a level of detail that is very hard to reverse.

Consider an example describing a simple management hierarchy. At best this is captured at the class diagram level by an arity constraint, although the more subtle aspects such as the required absence of cycles in the hierarchy graph can only be expressed by an informal annotation:

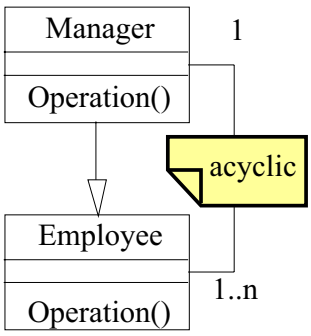

Figure 1: a manager/employee hierarchy

It can be seen that as we transit from the original requirements to the design, we replace abstract concepts by more concrete ones. The same holds for the implementation: in our simple example the arity constraint might be replaced by a precondition in a mutator function while the acyclicity constraint, if implemented at all, gives rise to some consistency maintenance code. On the other hand, we see that this decrease of abstraction is compensated by an increase in the level and amount of detail.

This observation holds in general and will be viewed as trivial by most software developers. However, we do well in analysing this transition from abstraction to detail as we descend through the various levels in the lifecycle of a software application. Typically, the amount of energy that needs to be applied increases with the level of detail; so does the need for technical skills. This generally makes an implementation artefact more valuable than a design artefact. Also, any ultimate defect in respecting the original requirements is detected at the lowest level, i.e. the implementation.

Initially, the development of a software application is achieved by this progression through the various abstractions: requirements, analysis, design and implementation.

However, once the implementation has reached the production stage, the tangible aspects of the prior stages are at best used as documentation in order to boost understanding of the actual code; at worst they become obsolete. This is a well-known phenomenon: under the pressure to bring software to market in the face of competition, or to correct flaws under the threat of contractual penalties, the management of evolving software all too often degenerates into updating implementations. Various directions have been explored to improve this situation: in general they imply some re- engineering activity applied to evolving implementations in order to extract abstractions and update e.g. design documentation. Hardly anyone uses an approach where design concerns drive the implementation process; programming environments that explicitly constrain the developer to design decisions are hard to find. Popular languages like Java evolve, but they evolve towards a more sophisticated type system: boosting genericity is a technical issue and hardly qualifies as support for e.g. architectural concerns.

It is our conjecture that during the development process, the concretisation of abstract concerns should not consist of some kind of erosion. On the contrary, any relevant feature should be kept available in any of the ulterior phases. We will in particular concentrate on the synchronisation between *design* and *implementation*. For the sake of this discussion we will discard requirements that cannot be expressed as explicit design directives. Our ambition is to augment an implementation such that it becomes a strict superset of its design; design can be extracted from an implementation by ignoring details; design can be interpreted by the programming environment and therefore enforced. We propose an approach called *Logic Meta Programming*, (or LMP for short) which will be described in the now following section.

Consider as an example a change in the manager/employee example where a decision to introduce *workforce pooling* results in the arity constraint to be changed into:

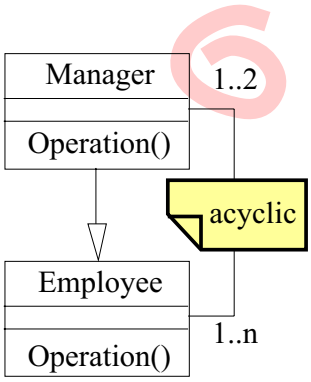

Figure 2: a manager/employee hierarchy with workforce pooling

The program code will probably need to be significantly changed with hardly an explicit link to the original arity constraint. We would prefer it to be explicitly present in the implementation as some kind of enforceable declaration, formatted in the proposed LMP-paradigm.

3. LOGIC META PROGRAMMING

Logic Meta Programming, or LMP for short, is the name we use for a particular flavour of multi-paradigm programming. The starting point for LMP is an existing programming

environment that is particularly suited for engineering large software systems. In this contribution, we have limited ourselves to a Java-based environment and to a Smalltalk-based environment. Next, we augment this environment by a declarative meta layer of a very particular nature. In the case of Java, i.e. a statically typed language, this meta layer might be implemented as a pre-processor or even an extension of the Java compiler itself. In the case of Smalltalk, it requires the addition of a number of classes to standard Smalltalk hierarchy. We are interested in a declarative approach; it seems intuitively clear that design information, and in particular architectural concerns, are best expressed as constraints or rules. Logic programming has long been identified as very suited to meta programming and language processing in general; see [DVD99] for related publications.

The acyclicity constraint from the manager-employee example on the previous page seems to indicate[2] the need for unification as an enforcement strategy. Anyway, we would like as much power on our side as possible, at least initially. We are not concerned with performance issues at this stage; neither do we intend to explore all avenues of declarative programming. For historical reasons, we concentrate on a Prolog-derivative for our logic meta language; its power and its capacity to support multi-way queries seem particularly attractive at this point. Finally, we make the symbiosis between the two paradigms explicit by allowing base-level programs to be expressed as terms, facts or rules in the meta-level; we will refer to this as a *representational mapping*.

```
class Array {
 private int[] contents;

 Array(int sz) {
 contents = new int[sz];
 }

 int getAt(int i) {
 return contents[i];
 }

 void setAt(int i, int e) {
 contents[i] = e;
 }
}
```

Figure 3a: a Java Array

```
Class(Array<?El>, {
 private ?El[] contents;

 Array<?El>(int sz) {
 contents = new ?El[sz];
 }

 ?El getAt(int i) {
 return contents[i];
 }

 void setAt(int i, ?El e) {
 contents[i] = e;
 }
})
```

Figure 3b: a generic Java Array

Consider the simple Java class in figure 3a: it implements an array of integers. Next to it in 3b the original class has been embedded in a meta-declaration using a representational mapping. The notation is fairly crude and to clarify it somewhat elements from the meta-program have been highlighted. Notice that the original element type was replaced by a logic variable.

This is an example of using LMP for code-generation; it was explored in [KDV98] under the exotic name *TyRuBa*. A proper query substituting `int` for **`?El`** in 3b would produce 3a. Actually, 3b is a simple example of how LMP can be used to introduce parametric types.

[2] Imagine for instance a tool to enumerate al cyclic calling graphs

A totally different way to view LMP is introduced in [RW98] as the *Smalltalk Open Unification Language* (SOUL). This approach actually applies constraints specified at the meta level to the base level program. The representational mapping is based on the presence of predicates that give access to syntactic elements belonging to the base level.

```
Rule transitive(?c1,?c2,?tried) if
      member(uses(?c1,?c2),?tried), !.
Rule transitive(?c1,?c2,?tried) if
      uses(?c1,?c2), !.
Rule transitive(?c1,?c2,?tried) if
      uses(?c1,?c3),
      transitive(?c3,?c2,<uses(?c1,?c3)|?tried>).

Rule cyclic(?c) if
      transitive(?c,?c,<>).

Rule uses(?c1,?c2) if
      class(?c1),
      method(?c1,?m),
      calls(?m,c3).
```

Figure 4: a cyclicity test

In the above example we assume the availability of predicates `class`, `method` and `calls` to access the structure of a base program. The `uses` rule establishes a transitive calling path between two classes, which allows us to derive the `cyclic` rule. This in turn could be used to enforce the acyclicity constraint from the manager-employee example.

4. *LMP* AND ASPECT ORIENTED PROGRAMMING

In [KDV98] a LMP framework is proposed that supports sophisticated type systems for statically typed programming languages such as Java. This framework, called *TyRuBa*, turns a type system into a computationally complete environment and allows a programmer to specify the static structure of a program as a set of logical propositions. In one of the next sections we will report on an experiment to use *TyRuBa* as a system to describe software architectures with. In this section we will build on the relationship between LMP and *Aspect Oriented Programming* (or AOP for short). We refer to [DVD99] for an extended bibliography; suffice it to say that AOP is concerned with the production of software as a result of a *weaving* process. The weaver is an AOP-related tool that is capable of merging aspects of a software application, each of them described in a specific aspect language.

In [DVD99] it is proposed that a LMP may well function as an aspect-oriented programming environment. As evidence for this, a well-known case for AOP (synchronisation of co-operating processes using an aspect language called COOL) is expressed in *TyRyBa*. An important conclusion from this experiment is the fact that a general-

purpose framework, in casu LMP, can be used to host aspect programs; hitherto, aspect languages were specific to the aspect under consideration.

In deference to the subject of this contribution, we will not concentrate on technical applications of AOP; instead we will consider an interesting application of AOP involving design as much as implementation. In [DD99] the idea is launched that *domain knowledge* might well constitute an aspect in the AOP sense. Separating some problem into its domain aspect and its implementation aspect by describing them in a some aspect language and then producing a piece of software by applying a weaver seems a very attractive approach.

Moreover, it seems to fit very well with the concept of software co-evolution introduced earlier on.

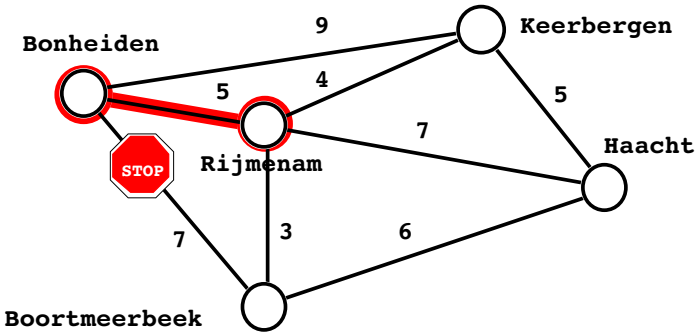

Figure 5: a shortest path problem

Figure 5 represents the test case proposed in [DD99] to explore this idea. The example is taken from a GIS-application, involving a mix of a conventional algorithm to compute a shortest path, and the specifics of the domain, which allow us to improve the basic algorithm.

```
branchAndBoundFrom: start to: stop
  | bound |
  bound := 999999999.
  self traverseBlock:
    [:node :sum |
     node free ifTrue:
       [sum < bound ifTrue:
          [node = stop
             ifTrue:  [bound := sum]
             ifFalse: [self branch: node sum: sum]]]].
  self traverseBlock value: start value: 0.
  ^bound
```

Figure 6: the branch-and-bound program

In order to keep things as simple as possible, we consider an elementary *branch-and-bound* strategy. In figure 6 this is implemented using an auxiliary `branch:` method in order to fix the sequence in which branches are selected.

```
branch: node sum: sum
  node free: false.
  node edges do:
    [:edge | self traverseBlock value: edge next
                                value: sum + edge distance].
  node free: true
```

Figure 7: fixing the selection order

Figure 7 contains a possible implementation for `branch:` and it contains an enumeration of all possible edges leaving a node. However, the message `edges` is no longer resolved by the base program, but by a query in the logic meta program containing the knowledge about this particular domain:

```
Fact city (Rijmenam)
Fact city (Boortmeerbeek)
...
Fact road (city (Rijmenam),city (Boortmeerbeek),[3])
Fact road (city (Keerbergen),city (Rijmenam),[4])
...
Fact prohibitedManoeuvre (city (Rijmenam),city (Bonheiden))
Rule roads (?current,?newResult)if
  findall (road (?current,?next,?distance),
  road (?current,?next,?distance),?result)
  privateRoads (?current,?result,?newResult)
Rule privateRoads (?current,?result,?newResult)if
  prohibitedManoeuvre (?current,?next),
  removeRoad (?result,road (?current,?next,?distance),?newResult)
Fact privateRoads (?current,?result,?result)
```

Figure 8: the domain knowledge

The particular flavour of LMP we use here is the *Smalltalk Open Unification Language* mentioned earlier on. The rule needed to compute the edges of a node would look something like this:

```
Rule edges(?node, ?result) if
     equals(?name, [?node name]),
     roads(city(?name), ?result).
```

In [DDW99] an explanation is given of how the base program and the meta program communicate. The basic idea is to effect a kind of linguistic symbiosis (see e.g. [PS94]) based on a two-way reification of language entities; in the case of SOUL this amounts to wrapping Smalltalk objects inside Prolog facts and vice versa. For example, the code `[?node name]` in the `edges` rule is reified Smalltalk code sending a unary message `name` to retrieve the name from the node currently associated with the variable; it returns a string representing the name. A number of technical issues need to be resolved still; in particular SOUL would seem

to lack in reflective power and needs to be extended with a number of reification operators. Also, the proposed test case would seem to border on the trivial. On the other hand, it shows that there is at least a lower bound to a category of problems that can be non-trivially decomposed into a *domain* part and an *implementation* part using AOP. An interesting research topic concerns the charting of this category and the development of tangible procedures to perform the related of decomposition. The proposed LMP approach seems at the very least attractive enough to function as a vehicle for this research.

5. *LMP* AND SOFTWARE ARCHITECTURES

Software architectures are concerned with the abstract structure of some software application in terms of building blocks, and the interaction between them. Starting from this fairly broad statement, a number of more specific —and sometimes competing—definitions have been proposed. In [KDH98] it is suggested that software building blocks need not be explicitly linked but may equally well be *classified*. Classification in its simplest form implies that all software entities are *tagged*; together with the possibility to *nest* classifications, this results in a very interesting view on architecture; its simplicity belies the power and expressiveness that was demonstrated in [KDH98].

In [MWD99] LMP is explored as a framework in which to express software architectures using this classification approach. In particular, virtual classifications are proposed, i.e. classification is not limited to simple tagging, but allows every software entity to be associated with a computational classifier. This could e.g. be a logical predicate that is evaluated every time a query is launched, its behaviour depending on values submitted by the query.

SOUL is proposed as both the target as the medium for this study: it is used as a kind of architectural description language and it is applied to the architecture of SOUL itself.

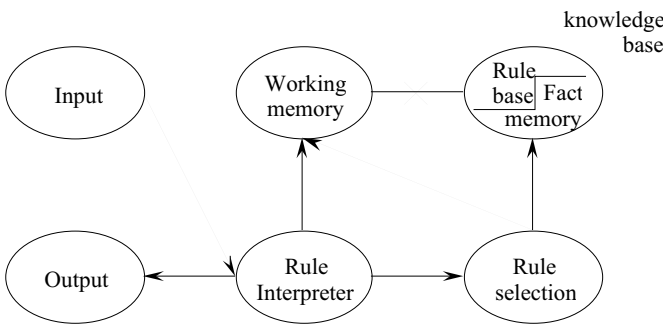

Figure 9: the SOUL rule base architecture

The kernel of SOUL is a logic query interpreter with the above architecture. This architecture is representative for rule bases in general; moreover it is sufficiently challenging to be used as a case study.

This architecture is obtained by means of classification. SOUL being implemented in Smalltalk, *methods* and *classes* are considered as building blocks, and classification is initially limited to a *uses* and *creates* relationship between these entities. For instance, the *working memory* is simple to define: it contains all classes that derive form a root class that specifies the generic structure of variable–value bindings. A more challenging example from [MWD99] is the rule that specifies how methods are classified as belonging to the query interpreter:

```
Rule methodIsClassifiedAs(?Method,queryInterpreter) if
      classImplements([SOULQuery],[#interpret:repository:],?M),
      reaches(?M,?Method).
```

Figure 10: a classifier rule

Bracketed terms are wrapped Smalltalk identifiers: `SOULQuery` is the class containing the `interpret:repository:` method that launches interpretation. The predicate *`reaches`* is similar to `transitive` in figure 4; it verifies that `?Method` belongs to the transitive closure of `?M` .

This classification crosscuts the static structure of the SOUL implementation and could not have been obtained through a simple hierarchical approach. It illustrates the true power of a computationally complete language in which to express an architecture in terms of classification.

In [MWD99] LMP is also used to express the connectors in figure 9 between the components defined by the various classifications. In fact, the *uses* and *creates* relationships are combined with *universal* and *existential* cardinality constraints to define a limited family of connectors; this results in a specification of the architecture of figure 9 as a ten–line SOUL fact. A definition for the *uses* and *creates* relationships for each of the kinds of implementation artefacts allows this fact to be used to perform *conformance checking* of any SOUL implementation.

This section described a second interesting experiment in using LMP to link the implementation of a software artefact to its design. Although the test case is small-scale and the sophistication of the connectors is limited, the results are promising. [MWD99] claim that it is possible to use this approach to define or even extract *architectural patterns*, which certainly illustrates the power of the proposed formalism. On the downside, performance is an issue requiring a lot of attention to make the LMP approach to software architectures a truly scalable one.

6. *LMP* AND SOFTWARE COMPONENTS

Software components and software architectures are notions that are strongly linked. However, this section is fundamentally different from the previous one. We are no longer interested in the declaration and enforcement of architectural rules and conventions, but in a *composer* environment. In this section we are much more tool-minded, and we want to investigate how LMP can help us drive the process of assembling components. This is of

course a major concern for everyone involved in software architectures; again we refer to [MJP99, MWD99] for a more comprehensive bibliography.

In [MJP99] an experimental generic builder tool is described and applied to the popular *Java Beans* component model. Its architecture is as follows:

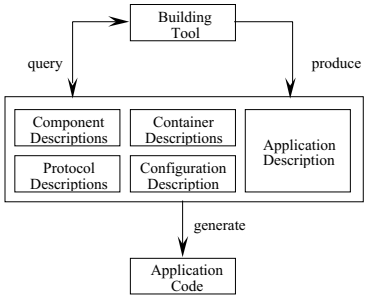

Figure 11: component builder architecture

The builder tool is supposed to guide a user in establishing a description of the application, and consequently generate the *Java Beans* application code. In order to do so, the tool must have access to a repository of descriptions of the various elements that constitute our component model.

The *TyRuBa* approach from [KDV98] to LMP is used to establish a set of facts and rules to define notions about *parts* and *containers*, and how to link them using *connectors*. Using these, the description of the application becomes a meta-program similar to figure 3b.

The component builder architecture provides the mechanism for describing *components*. A *Java Bean* for instance, is defined in terms of its properties, public methods and events. A set of facts allow the specification for accessor/mutator methods (in the case of properties), public methods and listener methods (in the case of events). To get a feeling for the way these are formatted, we include an example from [MJP99] that specifies the registration of an event listener:

```
Feature1(OurButton,
         method<void,add<Action<Listener>>,
         [Action<Listener>]>).
```

The expression bracketed by < > are compound terms and are used to support the representational mapping of Java in *TyRuBa*.

Next, it is necessary to describe *containers*; a container is a composite application (e.g. an *applet*) and it contains (generate) code to initialise the *parts*. A part is a description of how a component is used inside a container and it contains the specifics of the initialisation code. The *configuration* specifies how the writeable properties of parts belonging to a container are set. The *connection protocols* specify how the parts inside a container interact. We refer to [MJP99] for an extended example of the proposed component model architecture using *TyRuBa*; it would take up too much space to do so here.

The major contribution of [MJP99] is the proof (by construction) that it is possible to separate a builder tool from a component model. It is yet another illustration of the fact that

LMP can actively assist in expressing abstract and concrete aspect of a software application within the same framework. Although the proposed builder hardly qualifies as more than a prototype, it indicates an interesting avenue of research. In the spirit of Smalltalk's *Model View Controller*, sophisticated interactive tools can be seen to be separable in independent sections. However, contrary to MVC, a relatively simple framework approach is hardly ever sufficient and more sophisticated techniques are called for. It would seem that using LMP gives at least a partial reply to this concern.

7. CONCLUSION

This contribution is a first synthesis of work that has been going on in our lab these past couple of years related to declarative meta level programming. In particular, it covers several endeavours to marry a logic meta-program to a base program developed in a standard object-oriented programming language. In all cases the major concern was to effect a linguistic symbiosis in order to have the base program query the meta level to resolve issues at an abstract level, and to have the meta program access the structure of the base program.

This synthesis constitutes a push towards research in managing the co-evolution of design and implementation of software applications. We advocate the need to express design as closely integrated with the implementation and we propose *logic meta programming* as a possible way to effect a bi-directional link between the two. Our conjecture is that design becomes verifiable and possibly enforceable if it is properly expressed as a logic meta program. We explicitly address cases where software is subject to evolution, and where synchronisation between design and implementation is an issue. We are not only interested in the impact of a design change on the derived implementation; we are expressly concerned with the (unfortunately realistic) situation where an implementation is updated and the design needs to be brought in line with these changes.

Given the proper framework, a logic meta program expressing some design can assist a programming environment in constraining a programmer to abstract design rules that are visible at the level of the programming language only in terms of their constituent implementation details. An inkling that this might be feasible is given by the prototype *Java Beans* application builder.

Logic meta programming can also assist in separating the development of a software application into domain concerns and implementation concerns. Given that evolution of software at the implementation level is often inspired by implementation issues, this uncoupling of concerns might significantly and positively impact the proposed process of co-evolution. As was illustrated earlier on, the capacity of logic meta programming to express different aspect programs in one unifying framework seems far too attractive to ignore.

Possibly the most interesting direction described in this contribution is the management of co-evolution through virtual classifications. Logic meta programming seems extremely well suited to the annotation of object-oriented software with queries that are automatically triggered when some element is changed; depending on the content of the rule base, these queries can ensure the synchronisation between the various abstraction layers. Experiments with a simple tagging strategy have shown significant promise; opening up this strategy to query-based classification should give us the key to controlling the level of detail that co-evolution should respect.

There are of course a number of unresolved issues; this is after all a research topic barely out of the bud. A major concern is one of *efficiency* and *performance*. It would seem that depending on the degree of support offered by the proposed approach, various flavours of declarative meta programming with various performance ratings should be considered. Obviously, the vast field of research in declarative languages can function as an inspiration. Next, other levels in the lifecycle of software should be considered. In particular, expressing requirements (particularly the non-functional ones) in an LMP framework could prove to be a fascinating and rewarding research topic.

Finally, there is an enormous need to validate these ideas in a production setting. Although this has been initiated on a limited scale, many more experiments are needed. On the other hand, there is an even greater need for *comprehensive* software lifecycle management methods and tools. Steering co-evolution between design and implementation using logic meta programming seems to be an interesting step in this direction.

8. BIBLIOGRAPHY

[CL97] Documenting Reuse and Evolution with Reuse Contracts
Carine Lucas
PhD dissertation, Vrije Universiteit Brussel (1997)

[DD99] Is domain knowledge an aspect?
Maja D'Hondt and Theo D'Hondt
Proceedings of the ECOOP99 Aspect Oriented Programming Workshop (1999)

[DDW99] Using Reflective Programming to Describe Domain Knowledge as an Aspect
Maja D'Hondt, Wolfgang De Meuter and Roel Wuyts
Proceedings of GCSE '99 (1999)

[DVD99] Aspect-Oriented Logic Meta Programming
Kris De Volder and Theo D'Hondt
Proceedings of Reflection '99 (1999)

[JB99] Syntactic Abstractions for Logic Meta Programs, or vice-versa
Johan Brichau
Draft publication (1999)

[KDH98] A Novel Approach to Architectural Recovery in Evolving Object-Oriented Systems
Koen De Hondt
PhD dissertation, Vrije Universiteit Brussel (1998)

[KDV98] Type-Oriented Logic Meta Programming
Kris De Volder
PhD dissertation, Vrije Universiteit Brussel (1998)

[LD96] Bach and the Patterns of Invention
Laurence Dreyfus
Harvard University Press (1996)

[MJP99] Generic Component Architecture Using Meta-Level Protocol Descriptions
Maria Jose Presso
Master's dissertation, Vrije Universiteit Brussel (1999)

[MWD99] Declaratively Codifying Software Architectures Using Virtual Software Classifications
Kim Mens, Roel Wuyts and Theo D'Hondt
Proceeding of TOOLS Europe '99 (1999)

[PS94] Open Design of Object-Oriented Languages, a Foundation for Specialisable Reflective Language Frameworks
Patrick Steyaert
PhD dissertation, Vrije Universiteit Brussel (1994)

[RW98] Declarative reasoning about the structure of object-oriented systems
Roel Wuyts
Proceedings of TOOLS USA '98 (1998)

[TM99] A Formal Foundation for Object-Oriented Evolution
Tom Mens
PhD dissertation, Vrije Universiteit Brussel (1999)

[SLMD96] Reuse contracts: Managing the evolution of reusable assets
Patrick Steyaert, Carine Lucas, Kim Mens and Theo D'Hondt
Proceedings of OOPSLA, ACM SIGPLAN Notices number 31(10), pp. 268-285 (1996)