Model transformations in Converge

Laurence Tratt, Tony Clark
King's College London

September 8, 2003

Abstract

Model transformations are currently the focus of much interest and research due to the OMG's QVT initiative. Current proposals for model transformation languages can be divided into two main camps: those taking a 'declarative' approach, and those opting for an 'imperative' approach. In this paper we detail an imperative, meta-circular, object orientated, pattern matching programming language *Converge* which is enriched with features pioneered by the Icon programming language, amongst them: success/failure, generators and goal-directed evaluation. By presenting these features in a language suitable for representing models, we show that we are able to gain some of the advantages of declarative approaches in an imperative setting.

1 Introduction

Model transformations are currently the object of much interest and research. The Object Management Group (OMG), the standards body behind UML, recently published a Request for Proposals (RFP) named Queries Views Transformations (QVT) [OMG02] for model transformations which has bought to light an area of modelling technology that had hitherto been largely ignored. Model transformations are a vital constituent of the realization of the MDA vision [BG02]. Although there has been some discussion of the problem at hand [Béz01, dMES02] and some early attempts at tackling the problem [LB98b, LB98a, HJGP99, Gog00, LKM+02], surprisingly little progress has been made in tackling real-world transformations. The authors of this paper are members of the QVT-*Partners*, have contributed to the QVT-*Partners* submission to the QVT RFP [QVT03], and have also been co-authors on a follow up paper [ACR+03].

Current QVT submissions, and indeed existing approaches to model transformations in general, can be broadly categorized into two camps: those taking a 'declarative' approach, and those opting for an 'imperative' approach. The terms declarative and imperative can sometimes be rather contentious, and we use them with no small hesitation – they can also be rather crude mechanisms for classifying approaches. With that warning in mind, it is important to realize that in the wider context of programming languages there is a generally accepted consensus as to which of the two approaches most languages adhere to. Crudely put, a language is considered to be imperative if it has side effects and if it forces the programmer to be explicit about the sequence of steps to be taken when it is executed; languages that are side effect free and do not force the programmer to be explicit about the execution sequence are considered to be declarative. Declarative languages aim, to varying degrees, to allow the programmer

to express the desired result of an operation without having to express the complete low-level machinery necessary to achieve that result. Although this is a noble goal, declarative solutions can suffer from several problems in practice, including that: programs written in them may not always be executable; some real world problems are very difficult to express satisfactorily (many people are aware of the problems that input/output creates for most functional languages). Partially due to these problems, imperative languages continue to dominate the computing world.

The Icon programming language [GG96a] is a SNOBOL derivative, which contains several unique constructs which make it particularly well suited to the job of analyzing and transforming strings. Icon is notable in that whilst a cursory glance would suggest that it is a fairly standard imperative programming language, a closer examination reveals that the fundamental building blocks it is based on on are significantly different than those found in most other programming languages. The most important features for our purposes are the concepts of success and failure, generators and goal directed evaluation. Although Icon is an imperative language, and can be used in a way that is largely similar to other imperative languages, these features allow a programmer to express some very complex tasks in a manner that can be viewed as being more reminiscent of declarative approaches than traditional imperative approaches.

The fundamental idea behind our work has been to take the parts of Icon that make it well suited to string transformation and integrate that into an object orientated view of the world, particularly one that is amenable to manipulating models. The initial results of our work have resulted in a prototype of a language named Converge. The high level design goals of Converge are to merge the following elements into one whole: object pattern matching as found in the QVT-*Partners* QVT submission [QVT03]; the syntactic elegance and dynamic nature of Python [vR01]; the meta-circularity of ObjVLisp [BC87, Coi87]; and the features of Icon outlined above. Although the design and implementation of Converge are still in their relatively early stages, most of the fundamental concepts are in place and demonstrate that the approach is one that has much potential.

2 Model transformations

Put simply, the process of model transformation involves two models, one of which is a changed version of the other. In the context of Converge we are chiefly interested in transformation implementations – transformations which actually alter a model – as opposed to transformation specifications which check the result of a transformation for correctness. Transformations are increasingly recognized as a specialized, but highly important, task for which specialist tools, techniques and methodologies need to be developed.

The QVT RFP has given momentum to the until now rather hesitant work on model transformations, and thus any current model transformation work needs to be related to the QVT process. As the authors of this paper are members of the QVT-*Partners*, it is hardly surprising that Converge shares a number of features in common with that submission.

3 Converge

The high level design goals of Converge are to merge the following elements into one whole:

- Object pattern matching as found in the QVT-*Partners* QVT submission.
- The syntactic elegance and dynamic nature of Python.

  Whilst keeping this goal in mind, we have no desire to include wholesale some of Python's more obvious warts including: we replace Python's clunky scoping rules[1] with full statically determined lexical closures; we do not emulate Python's notion of bound and unbound class methods, which require unpleasant user-visible trickery to maintain in a meta-circular context.
- The meta-circularity of ObjVLisp.

  ObjVLisp has an elegant system whereby everything in a system is an object, and every object is an instance of a class (see section **??**) meaning that not only can metaclasses be created, but metaclasses are an equal part of the entire system. This allows powerful new abstractions to be built up by users.
- The salient features of Icon outlined above.

In this section, we first address some relevant background information required to understand the rest of the paper; then we describe features of Converge that are inherited from Icon; we then briefly cover the meta-circular nature of Converge influenced by ObjVLisp; and finally outline object pattern matching.

3.1 Icon

The Icon programming language, whose chief designer was Ralph Griswold, is a descendant of the SNOBOL series of programming languages – whose design team Griswold had been a part of – and SNOBOL's short-lived successor SL5. SNOBOL4 in particular was specifically designed for the task of string manipulation, but an unfortunate dichotomy between pattern matching and the rest of the language, and the more general problems encountered when trying to use it for more general programming issues ensured that, whilst successful, it never achieved mass acceptance; SL5 suffered from almost the opposite problem by having an over-generalized and unwieldy procedure mechanism. See Griswold and Griswold [GG93] for an insight into the process leading to Icon's conception. Since programs rarely manipulate strings in isolation, post-SL5 Griswold had as his aim to build a language which whilst being aimed at non-numeric manipulation also was usable as a general programming language. The eventual result of this aim was Icon [GG96a, GG96b]

As Converge imports the salient features of Icon almost wholesale, we do not go into specific details of Icon as these are effectively already covered by our description of Converge. We explain many of these features through the use of examples. Given Icon's decidedly left-of-centre approach to the task at hand, a complete explanation of these features is an ambitious task in the space available. Indeed in [GG93], Griswold states that when designing Icon 'there was a deliberate attempt not to copy from other

[1] The scoping rules in Python 2.3 and later are a substantial improvement on previous versions, but are still slightly hobbled by backwards compatibility.

languages or to develop refined versions of their features', a philosophy that led to the creation of unique language features, and that even subverts expectations such as 'array indexes start at 0' – in Icon they start at 1[2]. Interested readers should most definitely refer to [GG96a] for more details, as much of the description of Icon applies equally to Converge.

3.2 Syntax & semantics

3.2.1 Syntax

Converge's syntax is similar in style to that of Python and is thus indentation based. A standard `if` expression, for example, is written thus:

```
if a < b:
  sys.write("then clause")
else:
  sys.write("else clause")
```

Expressions are normally separated by newlines but can be concatenated with a semi-colon ';' on a single line with exactly the same meaning. Compound statements such as `if` can be written without newlines and indentation provided the meaning is still unambiguous e.g.:

```
if a < b: sys.write("then clause")
else: sys.write("else clause")
```

Converge is lexically scoped. The Converge compiler statically calculates variables scopes. Assignment introduces a new variable into a scope unless a `nonlocal x` compiler statement exists at any point in the block, in which case `x` refers to an outer scope (which are searched, in order, 'from inner to outer'). These rules are a significant deviation from those found in Python.

3.2.2 Datatypes

Converge has a number of simple, but powerful, datatypes as standards. Apart from the obvious strings and integers, lists, sets, associative arrays (a.k.a dictionaries, or hash tables) are represented as follows:

```
[1, "s", Dog]       // List of 3 items
{1, "s", Dog, 1}    // Set of 3 items
{"d" : 4, "k" : 11} // Associative array with 2 items
```

3.2.3 References

Icon exposes references (or 'pointers') to the user in several places, which creates some unexpected and unpleasant results, particularly with Icon's powerful but dangerous variable referencing. Failure to appreciate Icon's referencing rules, or the omission of the dereferencing operator '.' can lead to bizarre and difficult to trace bugs in Icon.

[2]Icon's indexing is also unusual as it refers to the position to the *left* of an item for positive indexes in order to make sense of list sections (perhaps more commonly known as list slices outside of Icon). Working from the other end of the list, negative indexes refer to the position to the *right* of an item with the addition that 0 refers to the position beyond the final list item.

In common with many modern programming languages, Python banishes references except to point to objects.

Converge follows the Python philosophy, and does some extra work internally to ensure that the language does what a human might reasonably expect, and not what is easiest for the compiler writer. This leads to a far safer and more uniform environment in which to program. For example, the following code fragment would print `2 2` in Icon, but prints `0 2` in Converge:

```
a := 0
Sys.write(a, " ", a := 2)
```

3.3 Implementation

Converge is a compiled language whose target is the Converge Virtual Machine (CVM). The compiler is largely standard. Many simple language constructs have direct equivalents in the CVM. Control structures however are largely created from a small set of primitives which are combined in various permutations to give the desired effect.

The CVM instruction set is, apart from object orientated related instructions, largely a subset of Icon's, with two notable differences that lessen the number if instructions in Converge. Firstly, most Converge operators are polymorphic (e.g. the Converge `+` operator is overloaded for integers, strings etc) as opposed to Icon which largely uses operators to operate on single types. Secondly, some things that in Icon are operators (e.g. list slicing) become method calls in Converge.

The CVM is slightly unusual in comparison to most virtual machines as function calls and so on are not implemented by recursion in the virtual machine. Internally the CVM deals exclusively in continuations, although these are not yet directly exposed to the user. This approach was taken in order to provide a simple mechanism for dealing with the concepts inherited from Icon. The approach taken in the CVM is similar to, but slightly more general than, that found in Icon. The internal use of continuations has some side benefits: it frees the Converge machine from being overly dependent on the stack size of its underlying implementation language (which is currently Python); it makes the possibility of optimizations such as tail-recursion easier.

3.4 Expressions, success and failure

Converge is an untyped expression based language. In identical fashion to Icon, Converge differs from most expression based languages whose expressions simply produces a value – Converge expressions either *succeed* or *fail*. If an expression succeeds it also produces a value. This simple concept is probably the most fundamental feature within Icon and Converge that makes it what it is, and is worthy of explanation and analysis.

Consider the following Converge snippet, which reads data in from an open file `file` and prints it to standard output:

```
file := Sys.file(file_name)
while line := file.readline():
  Sys.write(line)
```

At first glance, one could be forgiven for thinking that this snippet could have come from virtually any programming language influenced by the C/PASCAL schools of language design, albeit with marginally different syntax. There is in fact more going

on than meets the eye. The interesting goings on are in the conditional part of the `while` expression. What happens is that upon each iteration of the `while` loop, the expression `line := file.readline()` is evaluated. Evaluating this causes the expression `file.read()` to be evaluated. The `readline` function succeeds if there is still data left to be read in `file`, and if this is the case produces a string value. If `readline` succeeds, the value it produces is then assigned to the variable `line`, and the body of the while expression is evaluated with `line` set to this value. If the `readline` function failed, then it causes the assignment to the `line` variable to fail, which causes the `while` loop to terminate.

This method of execution is markedly different to the more explicit idiom found in most programming languages, and expressed here in pseudo code:

```
while (line := file.readline() && line != null) {
  stdout.write(line)
}
```

In this case the `readline` function equivalent returns a string if there is data left to be read in the file, and returns `null` otherwise. The programmer must explicitly check that the return value of the function is not null before proceeding with the body of the `while` loop.

Our Converge snippet is not yet fully idiomatic. By making use of the success and failure concept, as well as the fact that everything in Icon is an expression, the snippet can be more succinctly phrased thus:

```
while write(file.read_line())
```

This works exactly as before: if `read(line)` fails, then the `write` function will not be called, and will also fail, causing the `while` loop to terminate.

It is important not to think of the failure of an expression in Converge as being an exception. As Griswold explains, the failure of an expression is not intended to denote a catastrophic error, rather an expression should fail when a ‘a relation does not hold or if an operation cannot be performed but is not actually erroneous.’ [GG96a]

3.5 Generators

Another significant part of Converge is generators. Generators are expressions which can potentially produce more than one result. A simple example of a generator is `range(1, 10)` which generates the sequence of integers from 1 to 10 inclusive. Generators do not produce their values in one go: perhaps the easiest way to think of them is as being a way of implementing lazy programming in an imperative setting. A generator performs a `yield` expression which suspends the generator and returns a value to the caller; the generator then be *resumed* to potentially produce more values. Generator suspension is the computing equivalent of cryogenic freezing – all of the generators state (variable values, position in the program etc) is stored and restored automatically. When a generator has yielded all its values, it fails. Generators are therefore effectively a restricted form of coroutine [Knu97, Mar80].

As a simple example of generators, the following Converge snippet will print out all of the integers from 1 to 10 inclusive:

```
every Sys.write(range(1, 10))
```

This doesn't look very special at first glance, and a naïve assumption would be that code along the lines of the following would somehow expand to be internally equivalent to the following:

```
every Sys.write([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

In fact, the Converge code executes very differently internally. The `every` construct takes a generator and keeps reading values from it until it fails. On each iteration it will execute the body of the `every` construct, if any is specified (our example has none). In our example, the generator `range(1, 10)`[3] is thus created; straight away it succeeds and produces `1`, which is then printed out. The `every` construct then moves onto the next iteration, resumes the generator which succeeds and produces the value `2`. On the iteration after the generator has been resumed and yielded `10`, the generator will be resumed but will fail, and the `every` construct will terminate.

It is important to note that although all integers from 1 to 10 are printed as if they had been created as a single list, at no point in time does such a list exist in memory. Generators are thus particularly well suited to situations where large quantities of data need to be returned to a caller bit by bit, or where the data can be created on the fly.

A simple version of the `range` generator function can be written in Converge as follows:

```
func range(n, m):
  while n <= m:
    yield n
    n += 1
```

To print the integers from 1 to 10 is as simple as follows:

```
every Sys.write(range(1, 10))
```

Compare this to a more typical OO version, here written in Python, which has to maintain its state in instance variables:

```
class Range:
  def __init__(self, n, m):
    self.n = n
    self.m = m

  def hasnext(self):
    if self.n > self.m:
      return 0
    else:
      return 1

  def next(self):
    self.n += 1
    return self.n - 1
```

Instantiating the class, checking that it can produce another value and then actually obtaining the next value are as clunky as the definition of the class `Range`:

```
r = Range(1, 10)
while r.hasnext():
  print r.next()
```

[3] For the Python literate reader, the `range` function is akin to Python's `xrange`, not Python's `range`.

3.6 Goal-directed evaluation

If a generator is a part of an expression, then the failure of parts of an expression do not necessarily cause the entire expression to fail. Instead, Converge backtracks up to the most recent generator, resumes it to produce another value and then tries to evaluate the rest of the expression again; this effect is called goal-directed evaluation. Note that whilst superficially similar to features found in logic languages such as Prolog, goal-directed evaluation in Converge and Icon is unique because the sequence in which alternatives are tried is explicitly specified. Gudeman [Gud92] has a detailed explanation of goal-directed evaluation in general, with its main focus is on Icon, and presents a denotational semantics for Icon's goal-directed evaluation scheme. Proebsting [Pro97] and Danvy et al. [DGR01] both take subsets of Icon chosen for their relevance to goal-directed evaluation, compiling the fragments into various programming languages (Danvy et. al also specify their Icon subset with a monadic semantics); both papers provide good further reading on the topic.

The following example shows an expression which prints all factors of 3 between 1 and 1000:

```
every Sys.write(i := range(1, 1000) &
                pow(3, (range(1,sqrt(i)))) == i)
```

This example has two generators in it, one of which shows goal-directed evaluation. Working left to right through the expression, a generator successively produces the integers between 10 and 100, with the value produced assigned to `i`. `pow(3, (range(1, sqrt(i))))` is then evaluated, and for each value produced in the preceding generator, a test then takes place to see if it matches a power of 3. To evaluate this, Converge creates the generator `range(1, sqrt(i))`, reads the first value from that and thus tests if the expression `pow(3, 1) == i` succeeds. If the expression fails, the generator `range(1, sqrt(i))` is resumed to produce another value and so on. If this generator produces a value x which causes the expression `pow(3,` x`) == i` to succeed, then the overall expression succeeds, a number is printed out and Icon moves onto the next number in the $1 \ldots 1000$ sequence.

3.6.1 Bound expressions

Whilst backtracking is a useful feature of Converge, backtracking of the scale found in, say, Prolog is neither desirable nor feasible in an imperative setting. There is thus the concept of 'bound expressions'. A bound expression denotes a point at which backtracking stops. The most obvious point at which bound expressions occur is when expressions are separated by newlines in a Converge program. For example in the following code fragment the failure of the second line does not cause the first line to be re-evaluated:

```
a := file.read_line()
b := 1 == 2
```

Bound expressions occur in various other points. For example, each branch of an `if` expression is bound, which prevents the failure of a branch causing the entire `if` expression to be re-evaluated.

4 Metacircularity

Converge is a fully meta-circular and reflective language in the spirit of ObjVLisp. The essential idea is easy to state, but perhaps not so easy to understand (see [FD98] for further explanation): everything in the system is an object, and every object is an instance of a class. The system is bootstrapped with two classes `Class` and `Object`, with `Class` being a subclass of `Object`, and both `Object` and `Class` being instances of `Class`. Every object has a field `__of__` which refers to the class which the object is an instance of.

This scheme puts metaclasses on a completely equal footing with classes, because there is no such concept as 'a' metaclass – classes which inherit from `Class` are capable of playing the rôle of metaclasses, but there is no fundamental distinction between the two as found in e.g. Smalltalk [Coi87].

Although not relevant for many tasks, having metaclasses as first-class items is particularly relevant when dealing with modelling languages since metaclasses allow users to alter the behaviour of classes, thus facilitating a far more natural representation of various modelling language constructs within Converge.

Presenting a useful, readily understandable, example of a metaclass is something of a challenge. The following rather contrived example shows a metaclass which imbues its instances with an attribute `num_fields` which holds the number of fields the class had when it was initialized.

```
// As Field_Len is a subclass of Class, it can be used as a
// metaclass.

class Field_Len(Class):
  func __init__(name, supers, fields):
    super(name, supers, fields)
    self.num_fields := fields.len()


// Point is a class is an instance of the metaclass Field_Len
// instead of Class (which is the default metaclass if no
// other is specified). It is a subclass of Object which is
// the default superclass.

class Point of Field_Len:
  func __init__(self, x, y):
    self.x := x
    self.y := y


// The following line will print 2 (__init__, and to_str are
// inherited from Object).

Sys.write(Point.num_fields)

// Create an actual point

a_point := Point(20, 40)
```

Such an example hardly shows off the power of meta-classes, but does at least demonstrate that metaclasses in Converge are first-class entities. It also makes refer-

ence to the `__new__` and `__init__` methods: every object has an `__init__` method, but only subclasses of `Class` have a `__new__` method. `__new__` is responsible for creating objects, and filling fields with default values. `__init__` is responsible for fleshing a new object out.

5 Object pattern matching

Object pattern matching as found in the QVT-*Partners* submission is a powerful mechanism for succinctly expressing complex constraints on objects. It is analogous to textual regular expressions as found in e.g. Perl [WCO00], and to continue the analogy Perl support for object pattern matching is a built-in part of Converge. Compared to textual regular expressions, object pattern matching of this sort is relatively immature: Converge is, to some extent, a test bed for different ideas about object pattern matching in an imperative setting and thus we do not consider the current support for object pattern matching to be the final word on the subject. Pattern matching is, at the time of writing, also one of the more immature parts of the implementation.

The following code fragment will print `matched` if the object referred to by `d` matches the object pattern:

```
if m/(Dog)[name = "fido"]/(d): Sys.write("matched")
```

Object patterns are contained between forward slashes, and are first-class entities which can be called with parameters. The pattern can be preceded by various letters to indicate the particular flavour of object pattern matching to be used. `m` means 'match' which means that all parts of the object pattern must match successfully and exactly against the arguments given to the object pattern. The particular object pattern above will match successfully against an instance of the `Dog` class which contains a field named `name` which has the string value `"fido"`. If a potential object matches against these criteria than information such as whether the object is an instance of a subclass of `Dog`, or whether the object has extra fields are not relevant.

Patterns can contain free variables, which will bind to any value. For example:

```
if m/(Dog)[name = "fido", age = a]/(d):
  Sys.write("Matched against dog of age ", a)
```

This object pattern will match against a `Dog` object as before, but now requires the object to have an `age` field. Because the object pattern contains a single variable `a`, which has not been bound to a value previously, it will match against whatever the `age` field of the object contains and assign that value to `a`.

A searching object pattern can be expressed as follows:

```
if s/(Dog)[name = "fido", age = a]/({d, d2}):
  Sys.write("Matched against dog of age ", a)
```

This object pattern expects to be given sets as parameters, and it will try all the possibilities until it finds one which matches at which point it succeeds, which will cause object expression to succeed.

In general, one will not have a finite number of objects assigned to variables which can be manually fed to a searching object pattern. In this case generators can be fed as parameters, and via the Converge semantics the generators passed will be resumed until the overall expression succeeds. Imagine there is a generator function `dogs` which

successively returns all the dogs in the universe – we could evaluate this function to exhaustion, construct a set from the results and feed it to a searching object pattern, or we can use a generator as a parameter and a matching object pattern to print out the name of every dog whose age is 12:

```
every m/(Dog)[name = n, age = 12]/(dogs()):
  Sys.write("Matched against dog '", n, "'")
```

Object patterns themselves can also potentially be generators. For example the following searching object pattern will successively generate for every value in the input set which satisfies the pattern:

```
every s/(Dog)[name = n, age = 12]/({d, d2, d3, d4}):
  Sys.write("Matched against dog '", n, "'")
```

Patterns themselves can be considerably more exotic than those presented up until now. Set patterns are one such exotic form, and take the general form of:

$\{v_1$`, ... ,`v_2 `|` S_1`, ... ,` $S_m\}$

Set patterns are similar to set comprehensions in some programming languages. Values to the left of the vertical bar are matched as is; values to the right hand side of the bar must be sets, and are (possibly empty) subsets of the overall set. For example the set pattern {`"m" | S`} will match against a set which contains a string literal `"m"`, with any other values being assigned to the variable `S`. If the set `"m", "a", "b"` were to be matched against the preceding pattern, there is only a single outcome possible: the match would succeed and `S` would be assigned a new set `"a", "b"`.

With a set pattern such as {`v | S`}, one gets non-determinism in the matching. Given the set {`"m", "a", "b"`} to match against, the possible outcomes are as follows:

v	S
"m"	{"a", "b"}
"a"	{"m", "b"}
"b"	{"m", "a"}

Object patterns optionally have a `when` clause which must evaluate to true in order that the object pattern succeeds. For example, the following matching object pattern will only match against dogs over 5 years old:

```
every s/(Dog)[name = n, age = a] when a > 5/(d):
  Sys.write("Matched against dog '", n, "'")
```

Object pattern matching and generators are a powerful combination in a model transformation situation, as generators are a simple way to encapsulate complicated tree traversals that produce intermediate values. Given the simple model in figure 1 one can very easily write a function which traverses the model and successively returns all older generations from a given person:

```
func traverse_older(p):
  every parent := p.parents.generate():
    yield parent
    yield traverse_older(parent)
```

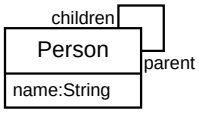

Figure 1: A simple model of families

Note how the `parents` attribute of `Person`, which according to figure 1 is a set, is turned into a generator which will successively return each of its elements by the `generate` method. Given this generic traversal function one can then write a simple object expression which will cause the name of every person who has two or more children to be printed[4]:

```
every m/(Person)[name = n, children = C]/ when \
        C.size() >= 2/(traverse_older(person)):
  Sys.write(n)
```

The astute reader will have noticed that this section is about pattern *matching*, with no corresponding constructs for object creation. See section **??** for further details.

6 Related work

The Unicon project[5] is in the reasonably advanced stages of extending Icon with object orientated features. It differs significantly from Converge in maintaining virtually 100% compatibility with Icon. Unicon's extensions to Icon, being effectively a bolt-on to the original, are not as natural a part of the language as one might wish.

One of Unicon's driving forces, Clinton L. Jeffery, is also partly responsible for Godiva[6], which aims to be a 'very high level dialect of Java' incorporating goal-directed evaluation. In reality, Godiva's claim to be a dialect of Java is slightly tenuous: whilst it shares some syntax, the semantics are substantially different.

Neither Unicon nor Godiva are meta-circular, and both are less dynamic languages than Converge. Nonetheless they both show that Icon's features have relevance in different areas.

7 Future work

Converge is very much in its early phases. There are many directions in which Converge could potentially move in that are yet to be fully explored. In this section we highlight a few of the more interesting issues we hope to explore further.

7.1 Scanning expressions

Unlike the previous aspects of Icon discussed in this paper, string-scanning expressions do not introduce a fundamentally different way of approaching programming. They are instead more of a syntactic convenience, albeit a significant one. A string-scanning expression looks initially like a truncated form of C's `?:` ternary operator:

[4]The backslash \ in the example means that the first two physical lines form one logical line in Converge.
[5]`http://unicon.sourceforge.net/`
[6]`http://www.cs.nmsu.edu/ jeffery/godiva/`

string `?` *expr*

Effectively this construct maintains two pseudo-global variables `&subject` – the string *`string`* being scanned – and `&pos` – the current position within the string – within *`expr`*. This saves the user from not only having to continuously write `func(`arg_1`, ..., `*`string`*`, `*`position`*`)` for a large number of standard functions, and also cuts down on the housekeeping associated with the maintenance of the current scanning position within the string: this allows the important aspects of the expression to shine through.

We wish to see how best string-scanning can be incorporated into an object orientated context.

7.2 Object creation

Whilst the QVT-*Partners* submission have object expressions which, whilst syntactically similar to object patterns, are capable of creating objects from scratch. For example the following object expression creates a new instance of the dog class:

```
(Dog)[name = "gido"]
```

The precise approach used in the QVT-*Partners* submission is not directly suited to Converge, because this circumvents the carefully constructed object creation and initialization phases that are a necessary part of Converge's metacircular nature. We would like to investigate the possibilities

7.3 Exceptions

As stated in section 3.4, failure and exceptions are two completely unrelated concepts. Converge currently lacks exceptions, which are a useful and important feature in modern languages. As the CVM is based on continuations, the introduction of exceptions should be relatively trivial, but it remains to be seen what the most sensible way to e.g. handle exceptions in the face of generators is.

7.4 Libraries

As experience with using Converge grows, we hope to start to be able to tease out common idioms of use, particularly those related to model transformations, into libraries. This will be vital if the approach is to scale up. Generators and metaclasses are two features that we feel are likely to make the creation of useful and flexible libraries a relatively painless process in Converge.

8 Imperative, Declarative?

In section 1 we noted that most current model transformation approaches opt for either fairly traditional declarative or imperative approaches. Experience would suggest that declarative approaches often incorporate some imperative-like features (e.g. so-called 'impure' functional languages such as ML [MTHM97]); however the converse does not seem to be as common. One of Converge's unusual features is that whilst clearly an imperative language, some of its features appear to have been influenced by, or are analogous to, those more commonly found in declarative languages. Having outlined these features we are now in a position to make concrete our original claim:

- The concepts of success and failure can be seen as being analogous to an implicit concept found in logic languages such as Prolog.
- Generators can be seen as a more explicit representation of lazy programming.
- Goal-directed evaluation is similar, but not identical, to Prolog's evaluation style.

9 Conclusions

We have outlined Converge, a modern compiled expression-based language based partly on Icon. We have shown how its unusual features make it particularly well suited to model transformations.

The authors would like to thank all of the QVT-*Partners*.

This work was partially funded by a grant from Tata Consultancy Services (TCS).

References

[ACR+03] Biju Appukuttan, Tony Clark, Sreedhar Reddy, Laurence Tratt, and R. Venkatesh. A model driven approach to model transformations. In *MDAFA 2003, Holland*, June 2003.

[BC87] Jean-Pierre Briot and Pierre Cointe. A uniform model for object-oriented languages using the class abstraction. In *Tenth International Join Conference on Artificial Intelligence*, pages 40–43, August 1987.

[Béz01] Jean Bézivin. From object composition to model transformation with the MDA. In *TOOLS 2001*, 2001.

[BG02] Jean Bézivin and Sébastien Gérard. A preliminary identification of MDA components. In *Generative Techniques in the context of Model Driven Architecture*, Nov 2002.

[Coi87] Pierre Cointe. Metaclasses are first class: the objvlisp model. In *Object Oriented Programming Systems Languages and Applications*, pages 156–162, October 1987.

[DGR01] Olivier Danvy, Bernd Grobauer, and Morten Rhiger. A unifying approach to goal-directed evaluation. *New Generation Computing*, 20(1):53–73, Nov 2001.

[dMES02] Miguel A. de Miguel, Daniel Exertier, and Serge Salicki. Specification of model transformations based on meta templates. In Jean Bézivin and Robert France, editors, *Workshop in Software Model Engineering*, 2002.

[FD98] Ira R. Forman and Scott H. Danforth. *Putting Metaclasses to Work: A New Dimension in Object-Oriented Programming*. Addison-Wesley, 1998.

[GG93] Ralph E. Griswold and Madge T. Griswold. History of the Icon programming language. *j-SIGPLAN*, 28(3):53–68, March 1993.

[GG96a] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language*. Peer-to-Peer Communications, third edition, 1996.

[GG96b] Ralph E. Griswold and Madge T. Griswold. *The Implementation of the Icon Programming Language*. Peer-to-Peer Communications, third edition, 1996.

[Gog00] Martin Gogolla. Graph transformations on the UML metamodel. In Jose D. P. Rolim, Andrei Z. Broder, Andrea Corradini, Roberto Gorrieri, Reiko Heckel, Juraj Hromkovic, Ugo Vaccaro, and Joe B. Wells, editors, *ICALP Workshop on Graph Transformations and Visual Modeling Techniques*, pages 359–371. Carleton Scientific, 2000.

[Gud92] David A. Gudeman. Denotational semantics of a goal-directed language. *ACM Transactions on Programming Languages and System*, 14(1):107–125, January 1992.

[HJGP99] Wai Ming Ho, Jean-Marc Jézéquel, Alain Le Guennec, and François Pennaneac'h. UMLAUT: An extendible UML transformation framework, 1999.

[Knu97] Donald E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, third edition, 1997.

[LB98a] Kevin Lano and J. Bicarregui. UML refinement and abstraction transformations. In *Second Workshop on Rigorous Object Oriented Methods: ROOM 2, Bradford*, May 1998.

[LB98b] Kevin Lano and Juan Bicarregui. Semantics and transformations for UML models. In Jean Bézivin and Pierre-Alain Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998*, pages 97–106, 1998.

[LKM+02] Tihamer Levendovszky, Gabor Karsai, Miklos Maroti, Akos Ledeczi, and Hassan Charaf. Model reuse with metamodel-based transformations. In Cristina Gacek, editor, *ICSR*, volume 2319 of *Lecture Notes in Computer Science*. Springer, 2002.

[Mar80] Christopher D. Marlin. *Coroutines: A Programming Methodology, a Language Design, and an Implementation*. Springer-Verlag, 1980.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The definition of standard ML*. MIT Press, 1997.

[OMG02] Object Management Group. *Request for Proposal: MOF 2.0 Query / Views / Transformations RFP*, 2002. OMG document `ad/2002-04-10`.

[Pro97] Todd A. Proebsting. Simple translation of goal-directed evaluation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–6, 1997.

[QVT03] QVT-Partners first revised submission to QVT RFP, August 2003. OMG document `ad/03-08-08`.

[vR01] Guido van Rossum. Python 2.2 reference manual, 2001. `http://www.python.org/doc/2.2/ref/ref.html`.

[WCO00] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly, third edition, 2000.