Using Icon-derived technologies to drive model transformations

Laurence Tratt, Tony Clark
King's College London, Strand, London, W2 3QH.

July 3, 2003

Abstract

Model transformations are currently the object of much interest and research. 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. The Icon programming language is a SNOBOL derivative which contains several unique constructs which make it particularly well suited to the job of analyzing and transforming strings. In this paper we discuss model transformations, analyze the relevant parts of Icon that lend themselves to transforming strings, and then propose how some of Icon's unique features could be incorporated into a model transformation approach that partially blurs the distinction between 'declarative' and 'imperative' approaches.

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 and have contributed to a submission to the QVT RFP [QVT03] and who have also been co-authors on a follow up paper [ACR+03].

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 quick glance would suggest it is a fairly standard imperative programming language, upon closer examination it becomes apparent 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 scanning expressions; see section 3 for details of these features.

In this paper we discuss model transformations in general, analyze those features of Icon which make it well suited to transforming strings, and then propose how some

of these features could be incorporated into model transformations. Following observations that current approaches to model transformations fall into one of two distinct camps – crudely categorized as being of the declarative or imperative schools of thought – we propose that a model transformation language influenced by Icon can blur some of the distinctions that currently exist.

2 Model transformations

Put simply, the process of model transformation involves two models, one of which is a changed version of the other. In this paper 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. In this section we give an overview of QVT, go into a little more detail on the two main philosophical approaches currently being proposed for QVT.

2.1 QVT

The QVT process is relevant as it has provided a focus for those developing model transformation technologies; in section 2.2 we discuss one of the defining differences between the various approaches that are currently being explored as solutions to the RFP.

2.2 Declarative and imperative approaches

The model transformation proposals submitted to the QVT RFP can be broadly categorized as being either declarative or imperative in nature. 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. Most languages can be easily categorized using this definition – the difficulty comes when languages place themselves close to the dividing line between the two camps. To give real world examples, the C programming language is universally agreed to be an imperative language, and Prolog to be a declarative language. Straying into the transformation world, textual regular expressions as found in e.g. Perl are declarative. XSLT [W3C99] on the other hand is a much harder beast to classify. Its use of XPATH patterns and the fact that it is side-effect free would seem to easily classify it as declarative language; however XSLT is, overall, rather procedural [BMN02]. Although XSLT is generally regarded as being declarative, one could construct a fairly convincing argument that when stripped of its peculiar syntax (and

taking into consideration its somewhat convoluted operation), it could be considered imperative. In our context XSLT is an example of the dangers of crude categorization.

'Declarative' is an umbrella term for an entire array of very different approaches: the most common styles of declarative languages are functional languages (e.g. Haskell) and logic languages (e.g. Prolog). Some of the QVT submissions have opted for a different style – constraint solving. Constraint solving is a relatively new area of research, and is substantially different from the 'standard' declarative approaches. The most obvious difference that this makes is that whereas 'standard' declarative languages are executable, constraint solving introduces significant challenges in relation to executability. We are not overly concerned about constraint solving in this paper as, whilst a fascinating area of research, it is substantially different from more traditional approaches, and brings its own set set of largely unresolved issues.

Arguing the case for either declarative or imperative approaches can lead one onto dangerous ground. In this paper we do not argue for one approach over the other – what we aim to do is to take some of the benefits of a declarative approach into an imperative setting.

3 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] which came into life in the late 70's and is still in active development to this day.

In this section we detail[1] what we believe to be the most relevant aspects of Icon, and explain them partly through the use of examples. We believe that these features have proven themselves in the real world and that useful lessons can be learned from them. Given Icon's decidedly left-of-centre approach to the task at hand this is a surprisingly 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 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.

[1] In this position paper the full details are elided in the interests of brevity.

[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.

3.1 Expressions, success and failure

Icon is an untyped expression based language. However it is unusual in that whereas in most expression based languages every expression produces a value, Icon 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 that makes it what it is.

3.2 Generators

Another significant part of Icon is generators. Generators are expressions which can potentially produce more than one result. A simple example of a generator is `1 to 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 `suspends` itself to yield a value, and can then be *resumed* to potentially produce more values. They are therefore effectively a restricted form of coroutine [Knu97, Mar80].

3.3 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, Icon 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 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.

3.4 String-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, as by maintaining two pseudo-global variables `&subject` – the string *`string`* being scanned – and `&pos` – the current position within the string – within *`expr`* they free the user from a couple of common headaches: having to continuously write `func(`arg_1`, ..., `*`string`*`, `*`position`*`)` for a large number of standard functions; manual maintenance of the current scanning position within the string. Note that `&subject` and `&pos` aren't global variables in the traditional and often abhorred sense of the word – for example, nested string-scanning expressions properly preserve the correct values of the variables.

3.5 Summary

Icon has a reputation for allowing complex string scanning expressions in an easy and succinct manner, whilst also being usable as a general programming language. Although in the world of string processing 'transformation' is perhaps thought of as a rather grand term for an everyday task, Icon also excels at transforming strings. By unburdening the programmer from having to manually create a large amount of tedious machinery to analyze and process a string, Icon also puts some distance between

itself and most other imperative programming languages. The combination of string-scanning expressions, goal-directed evaluation, generators and the success/failure concept allow a programmer to concentrate much more on *what* they want a transformation to do, freeing them from having to specify as much of the *how* as might otherwise be the case. But at the same time, Icon is still most definitely an imperative language, and thus maintains the benefits, and familiarity, of the imperative approach.

4 Model transformations and Icon: putting it together

In section 2 we outlined the area of model transformations; in section 3 we outlined some of Icon's most distinctive and useful features. This section is intended to explain how we believe Icon's features can be adapted to, and useful in, the model transformation world.

In section 2.2 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 Icon'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. In particular:

- 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.

The question is thus what use can we make of these sorts of features in model transformations?

The aim of this paper is, having identified useful features from the Icon programming language, to outline a model transformation system where these features are incorporated into. There are several challenges in this including:

- Icon is aimed at string processing and other linear data structures, whereas models are best represented as graphs. The assumption of linear data structures is threaded through many areas of Icon, from the language itself to library functions.
- Icon is chiefly aimed at operating on immutable data structures. Again, a fundamental assumption built into many areas of Icon is that the linear data structures over which it operates are immutable. It is often desirable that model transformations update models in place, and thus this assumption of immutability no longer holds.
- Icon has only limited facilities for composing transformations. We wish to be able to compose model transformations in more sophisticated ways.

We believe that the result of integrating Icon-like features into a model transformation approach will be more advanced transformation languages and libraries, which

will allow transformations to be expressed accurately and concisely, and to be created more quickly than existing approaches.

The first half of the full paper will be structured in a similar manner to this position paper (but with extended analysis and examples of Icon's relevant features); the second half will be a discussion and analysis of a small indicative model transformation language that incorporates Icon-esque features.

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.

[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.

[BMN02] Geert Jan Bex, Sebastian Maneth, and Frank Neven. A formal model for an expressive fragment of XSLT. *Information Systems*, 28(1):21–39, 2002.

[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.

[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`.

[QVT03] QVT-Partners initial submission to QVT RFP, 2003. OMG document `ad/03-03-27`.

[W3C99] W3C. *XSL Transformations (XSLT)*, 1999. `http://www.w3.org/TR/xslt`.