In Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centgric Computing, Corvallis, OR, USA

A Domain-Specific Visual Language for Report Writing Using Microsoft DSL Tools

Ruskin Dantra[1], John Grundy[2] and John Hosking[3]
Prism New Zealand[1], Dept. of Electrical and Computer Engineering[2] and Dept. of Computer Science[3]
University of Auckland
Auckland, New Zealand
ruskin.dantra@gmail.com[1], {john-g[2], john[3]}@cs.auckland.ac.nz

Abstract

Many domain specific textual languages have been developed for generating complex reports. These are challenging for novice users to learn, understand and use. We describe our work developing the prototype of a new visual language tool for a company to augment their textual report writing language. We describe key motivations for our visual language tool solution, its architecture, design and development using Microsoft DSL tools, and its evaluation by end-users.

1. Introduction

Enterprise tasks and processes tend to be complicated and require considerable end-user training and support. Writing reporting scripts to query databases and display the extracted information is a good example of this. Industrial databases are complex and hold large amounts of data, the majority of which is irrelevant to the user. They also have unfriendly features such as arcane table naming, implicit relationships and cascading associations [*1*].

While there are many existing report writer approaches [*2*] [*1*] [*3*] [*4*], these are typically low level in their approach. Generic reporting tools often do not cater for the exact needs of an end-user for a particular enterprise database. This forces an enterprise (organization) to introduce their own proprietary textual domain specific languages (DSLs) which require significant programming ability on behalf of the report designers. Issues in using such languages include arcane syntax, hidden dependencies and a lack of support tools making the languages difficult for novice users thus decreasing their productively [*1*] [*5*] [*6*].

This paper describes the development of a visual domain specific visual language (DSVL) and an environment that assists end-users to rapidly design and implement reports sourced from an enterprise system and its database for the printing and graphics art industry. This is done by creating a meta-modeling framework which also in turn assists developers to modify and add new elements to this reporting language.

The language and environment replaces an existing textual domain specific language providing a more accessible and productive approach to report writing for novice and intermediate end-users. Our contributions are two-fold: the design, prototyping and evaluation of a visual report writer language and support tool; and as an example of industrial usage of a domain-specific visual language and meta-tool development approach.

We begin by describing the domain of the target enterprise system and its existing textual report writing language. During the description we will also highlight its inherent issues which motivated our new approach. This is followed by a description of our new DSVL, environment and framework. Following this is case study of its use to clearly demonstrate the improvements a given DSVL has over its textual counterpart.

We then describe the details of the DSVL implementation, using Microsoft DSL Tools [*7*]. We then describe the evaluation methods and present its results. The paper concludes with a brief set of conclusions and possible future work.

2. Background and Motivation

Prism WIN is a fully integrated and configurable enterprise Management Information System (MIS) for the printing and graphics arts industry [*8*]. Prism WIN has a number of integrated modules for estimation, inventory management, production management, financial reconciliation, sales ordering and processing and facilities management. The Prism report writer language (RWL) is an interpreted textual DSL that allows Prism WIN end-users to write powerful data mining queries and present the results graphically as reports [*9*].

Figure 1 shows a textual RWL script for a report that extracts the customers from a Prism database and

prints out their active jobs, i.e. those that are beyond quoting.

```
1.  Code CASE_STUDY_1
2.  Type Standard
3.  Access STSR
4.
5.  Scan RM
6.          Print RM_CUST + RM_NAME;
7.          Print "All Jobs For " + RM_NAME;
8.          Scan QM
9.              Choose (QM_CUST_CODE, MATCH, RM_CUST)
10.             Choose(QM_QUOTE_JOB, MATCH, QMM_JOB)
11.
12.             Print QM_JOB_NUM + QM_TITLE;
13.         End
14. End
15. Print StandarReportFooter;
```

Figure 1. RWL script example

Lines 1 to 3 are the header definition (*ControlLine*), which provide metadata for searching and indexing this particular report through the Prism WIN MIS. Line 5 is a *Scan*. This loops through the specified Prism WIN database view (*RM*), printing out customer information. This is followed by an inner *Scan* which loops through the QM database view selecting active jobs for the current customer in the outer Scan and prints out information for each customer-job record. A *Scan* can be simply seen as a *Select* statement within the Structured Query Language (SQL) and the *Choose* statements allow two *Scans* to be joined (inner join). For this example the first *Choose* joins the *RM* and *QM* views and the second *Choose* only selects *Jobs* (not quotes).

The *Scan* is followed by a standard footer which simply prints any arbitrary (configurable) text after the end of the report.

RWL also has variables, including clumps which are aggregates of arbitrary objects (such as database columns and/or other variables), report headers and footers and many inbuilt functions. Prism has also developed an IDE for RWL, Prism Scribe.

This provides wizard support for common templates, libraries of metadata, functions and code snippets, syntax colouring and completion. This IDE is, however, all hard coded in VB6 making syntax and environment changes difficult.

As the current IDE was incompatible with Microsoft Vista and changes were also desired to extend RWL's functionality, Prism was motivated to explore alternative approaches to report specification. In particular, Prism wished to explore a more model driven approach, based on a comprehensive meta-model for RWL specification and a compatible DSVL and environment for model driven report specification.

Therefore the key enterprise requirements arising are to allow end-users to query their complex database schema to mine information and to allow the end-user to lay this information out as they need. Secondary requirement was also to allow developers to roll out new reporting features with ease. These enterprise requirements were translated to the following technical requirements:

- An extensible meta-model that correctly represents an abstraction of RWL and its semantics
- An extensible DSVL and environment that allows end-users to easily specify and test Prism reports. This should use an end-user accessible metaphor, validate RWL semantics and constraints, provide simple access to required database meta-data, support versioning and test cases and afford modularity and scalability
- A code generator that generates RWL reports from DSVL specifications
- Delivery using a widely used platform, such as Microsoft Visual Studio or Eclipse

Many report writing tools have been developed. These include Crystal Reports [*10*], Visual FoxPro report writer [*3*], Eclipse business intelligence tools, and Visual Studio report designer [*11*]. These approaches use a combination of wizard-driven templates and/or textual domain-specific visual languages. Some, such as Crystal reports, provide a limited visual report layout designer and wizards to set up data sources. However most are generally designed for programmers or experienced data modellers and being general-purpose are not Prism-specific tools. Moreover Prism RWL offers integrated support with the Prism WIN MIS which other report writing languages may offer with a strenuous API or not offer one at all.

A range of other approaches have been developed to provide more end-user friendly report writing tools. FastReport Studio [*12*] provides visual layout and visual query specification for any ODBC source. As mentioned earlier this is also another general purpose solution and not specific to Prism. Customization is likely to be difficult and the software itself is intended for end-users with some programming knowledge. Report Sharp Shooter [*13*] provides a visual layout designer, report wizard, reusable templates with cascading structures, an accessible DOM-based model, preview and export. It is also intended for programmers.

FoxQ [*5*] uses a form-based approach to specifying XQuery XML document queries, allowing high-level authoring of reports. This is intended for document processing rather than database query and reporting [*5*].

A number of visual query languages have been developed, some with reporting capabilities. GQL [*14*]

provides a general purpose graphical query language, though not oriented to business reporting.

A WebMLbased visual language tool [*15*] provides GIS-oriented database querying and reporting though not business oriented reporting. HyperFlow [*16*] provides general purpose end-user oriented information analysis support with a visual query metaphor.

IViz uses a dataflow visual language to synthesize complex graphic reports from data [*17*]. Co-ordinated Queries [*4*] and web server page generators [*6*] provide visual abstractions for specifying both queries and contents of reports. Neither is oriented towards business reporting per se and neither to Prism specific databases.

3. Our Approach

We determined that we would need to design a domain-specific visual reporting language for Prism and build a prototype tool to support it. The extensibility requirements, however, meant that we had two target end-user groups to consider: Prism developers who would maintain and extend the meta-model, DSVL and environment; and end-user report authors. Our approach involved four core steps:

1. Designing a visual RWL meta-model, providing a representation of the core modelling elements and inter-relationships of the textual RWL
2. Specifying RWL constraints on the meta-model: this provides the main semantics of the language
3. Exposing the meta-model to report authors via a visual language: this involved designing a suitable visual language oriented towards end-user report writer authoring and building a visual modelling tool using this language to populate instances of the meta-model while adhering to its constraints
4. Designing and deploying code generators to generate textual RWL script from the visual RWL model: we decided to generate textual RWL scripts from our DSVL so that we could leverage the existing complex Prism report generation engine.

Several approaches were possible: (a) a WYSIWYG report designer that allows end-users to manipulate something close to the final appearance of their report, like MS Word for documents; (b) an abstract representation of report layout e.g. broken into headers/body/footers; (c) a visual representation of Prism textual report script elements but making explicit their relationships; or (d) a dataflow-like language describing data sourcing, transformation and output. As mentioned before all of these approaches have trade-offs in terms of ease of use, closeness of mapping to the problem domain, expressiveness, scalability and ease of implementation.

Conceptually a WYSIWYG approach is appealing [*5*]. However, as reports have complex repeating groups, database queries and formulaic translations of data for output, providing such an approach limits expressiveness and is very challenging to implement. Abstract report layout e.g. as header/footer/body/group bands, is quite common [*11*] [*3*] [*13*]. However, this approach suffers from introducing many hidden dependencies and can be very *"viscous*" i.e. hard for end-users to change report designs. The dataflow metaphor appears attractive and has been used in a number of visualization and/or reporting-oriented approaches [*17*] [*18*]. However these rapidly become very complex as queries and transformations grow. After surveying target end-users with mock-up design examples we chose to use option (c) above: developing a new visual language based on existing Prism textual report scripting language constructs, analogous to [*14*] [*4*]. This provides existing reporting abstractions with most dependencies made explicit via relationships; grouping of hierarchical constructs; and database element and report element constructs explicitly linked. We also borrowed the *"banding"* concepts from approach (b) but use the "swim lane" metaphor from activity diagrams to realize this. We found this easier and more convenient for end-users than rigid report bands and layout.

We tried two variations of approach (c), which are shown in Figure 2 and

Figure 3 below.

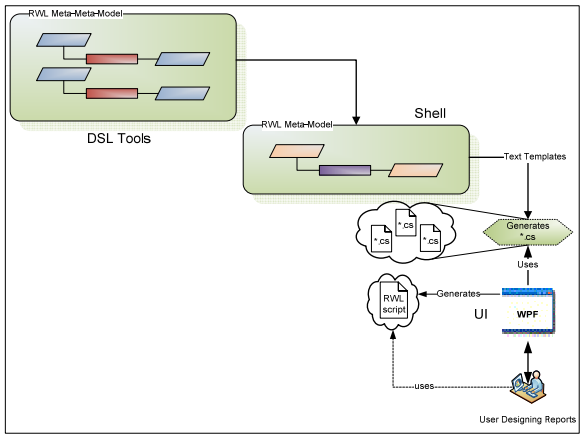

Figure 2. Approach one

Figure 3 shows an outline of the approach we took. The approach shown in Figure 2 was discarded in the early design stages due to the overhead of

implementing a whole new user interface to accentuate the RWL meta-model as Microsoft DSL Tools already offered an integrated shell to host visual languages. A trade-off had to be made between flexibility and a rich user interface experience which was offered by our first approach against rapid prototyping which was offered by the second.

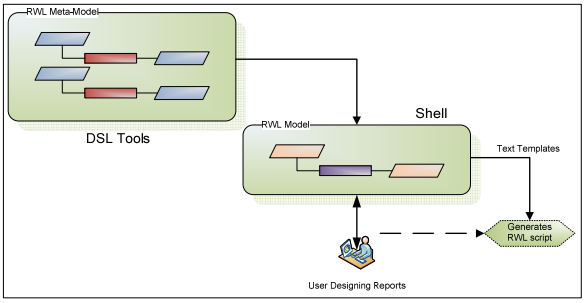

Figure 3. Approach two

Therefore our approach (shown in
Figure 3) comprised of three core phases, each phase mapped to one or more steps enlisted earlier in this section.

1. Design the meta-model of the RWL representing as much information as possible (mapped to Step 1)
2. Constraints which cannot be represented using the meta-model were encoded within a rules engine via code (mapped to Step 2) [*9*]
   E.g. Mandatory fields within a meta-model element;
   Scan must be mapped to a database view
3. Developed a formal visual notation and tool for our meta-model which can be used by end-users. The tool will be used to generate the required RWL script from the corresponding visual RWL model (mapped to Step 3 and 4)

4. Usage Example

We briefly illustrate end-user report specification using our visual RWL tool. Initially the user creates a new report or opens an existing report. Figure 4 shows the end-user interface while beginning design of a new Prism report. A palette (1) provides available report elements, including header, body, footer, variables, control lines, scans, pages, comments, literals, functions, clumps, etc. We chose to use a set of "swim lanes" (2) to group report elements into key sections including Header, Body, Footer, and Variables, note that the orientation of the swim lanes can be easily altered if needed. A property sheet (3) provides context-sensitive element properties for editing. For example, the properties for the ControlLine1 element (4) are shown.

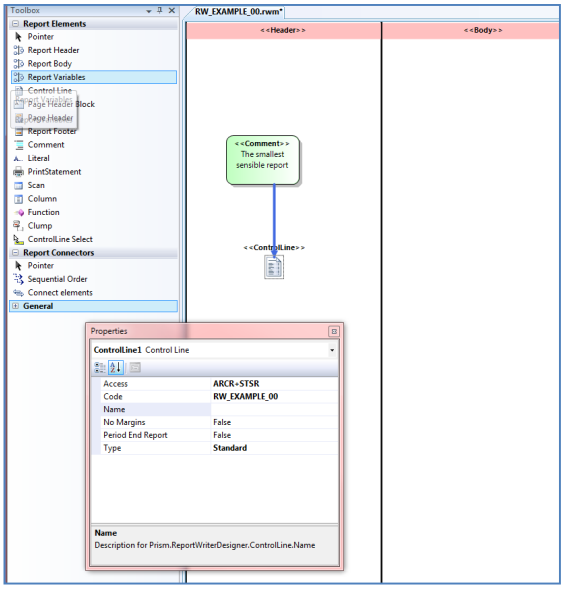

Figure 4. Visual report writer tool user interface

In this example the user has added a header and body element by drag-and-drop from the tool palette, an initial ControlLine to the header and initial Scan to the body of the report. Properties may be set when an element is added or later when selected by the user.

One of the more complex report elements supported by the Prism report writer language is the *"Scan"*, or query from a Prism database table. In Figure 6 a *Scan* element is added by the user to query a Prism database table for data items to add to the report. The user needs drags-and-drops the *Scan* from the palette (Figure 4 (1)) onto the canvas and then selects the table on which to perform the *Scan*. This can be followed by adding any conditional statements (*Choose*) which can be done by a context menu (Figure 6 left) followed by selecting the conditions (Figure 6 right). Other report elements can be added using a similar approach of drag-and-drop; link to existing elements; or specify properties. The tool enforces underlying RWL meta-model semantic constraints e.g. you cannot add clump outside a scan; you cannot add a scan to a header; and so on.

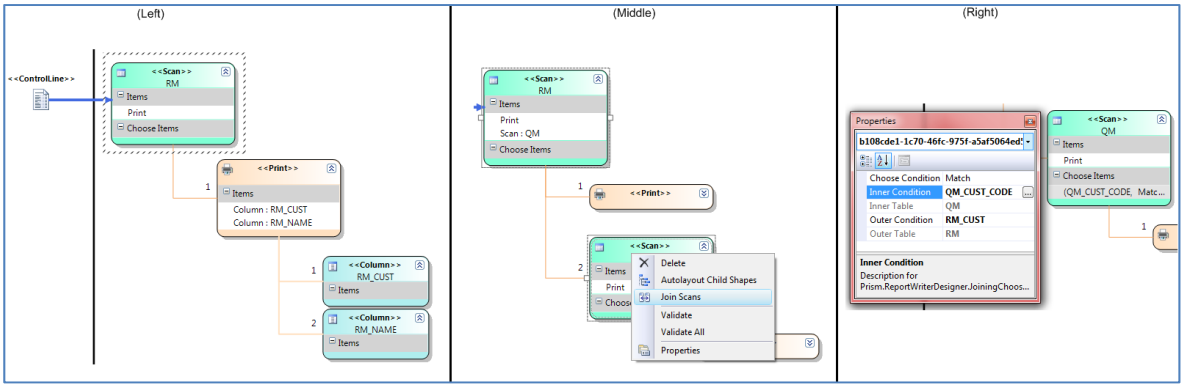

Figure 5. Design evolution while designing the report

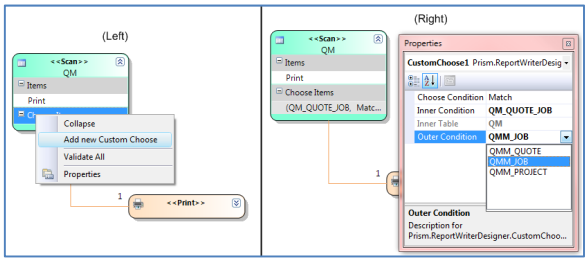

Figure 6. Adding and specifying a Scan

The user continues to add report elements and evolve the design: e.g. report footer; page header/footer; print statements, further scans; scan elements; literals and so on.

For example, in Figure 5 (left) the report has been extended to include two print statements (that generate report body content); *Scans* (selections) from the Prism RM and QM tables (middle); table field items to appear on the printed form. Sequencing can be specified for report items to enforce ordering of printed items on the final report (Figure 5 (left), arrowhead going from *ControlLine* to *Scan*). Figure 5 (middle) also shows how two Scans can be automatically joined on their mapping column automatically. This join is configurable by the user after it is added; Figure 5 (right).

To manage complexity of reports elision and additional visual report writer diagrams are supported. The user can collapse/expand items and their sub-items on the report. They can also create additional diagrams to specify parts of the report and package these via parameterization for reuse. Figure 5 (middle) (and Figure 6 (left)) shows an example of a collapsed *Print* statement. The user can generate a textual Prism report script from the visual report writer language. Figure 7 shows the Prism report script generated from the example in Figure 6. This report script can then be run by Prism report engine against the Prism ERP database to generate a report to a file. The resulting report file is then opened and report content shown to the user.

```
//---------------------------------------------------------------------------
// <auto-generated>
//     This code was generated by a tool.
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
//
//     Generated on 6/06/2009 1:35:16 p.m.
//     ToolVersion 1.0.0
// </auto-generated>
//---------------------------------------------------------------------------

Code        RW_EXAMPLE_03
Type        Standard
Access      STSR
Name        "Report Writer Example 03"


Scan RM
    Print RM_CUST + RM_NAME;
    Scan QM
        Choose (QM_CUST_CODE, Match, RM_CUST)

        Print QM_TITLE;
    End
End
```

Figure 7. Generated textual RWL report

5. Architecture and Implementation

Prism's requirements motivated a model-driven engineering approach to the visual tool development allowing us to capture the existing textual RWL concepts and rapidly prototype a tool [*19*] [*20*]. We chose to use the Microsoft Domain-Specific Language (DSL) tools platform to design and build our prototype [*21*] [*7*]. Other meta-tool platforms, such as MetaEdit+ [*22*], Eclipse GMF and our own Marama Eclipse-based platform [*23*] were possible choices. We chose to use Microsoft DSL tools as Prism desired a Microsoft-based solution and we believed it provided a suitable platform for robust visual report writer delivery to end-users. To realise our visual report writer prototype we followed the process outlined in Section 3:

- Meta-model development: we built and refactored over time a large meta-model to capture all of the commonly-used Prism textual reporting language elements and many of the more obscure ones. Part of this meta-model is shown in Figure 8. We designed our meta-model to be as extensible as possible; one of the major attractions of the DSL tools approach is that Prism developers can

extend this meta-model to add new visual report writer features easily.

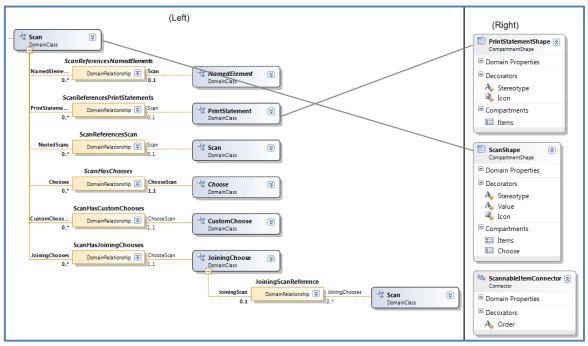

Figure 8. RWL meta-model

- Specify constraints on the meta-model: we used the DSL tools constraint specification facilities to constrain the semantics of our visual reporting tool. We also used where necessary C# functions and custom property sheets and choosers. These allowed us to implement e.g. meta-data querying facilities to the existing Prism database. This ensures end-users only use valid tables, fields and types in their reports.
- Visual report writing language: we used the DSL tools facilities to specify visual forms of meta-model elements and relationships i.e. icons and connectors to represent meta-model elements in the visual report writer diagrams. Each meta-model element has a visual form, as shown by the examples in Figure 8 (right). We used DSL tools visual editor constraints to provide layout capabilities including swim lanes, collapsing and expanding, and automatic layout of sub-items for nested elements such as the *Print* and *Scan* model elements.
- Code generator: we wanted to use the existing Prism reporting engine to actually interpret and run reports specified in our visual report writing language. The easiest way to achieve this was to generate textual report script and run it via the Prism reporting engine. We used the T4 templating engine within the DSL tools to synthesise textual report scripts from the report model and show the user the result which can then be importing and run via the Prism MIS.

6. Evaluation

We undertook two complementary evaluations of our solution. The first was a continuous design evaluation. This used the Cognitive Dimensions (CD) framework [*23*] during each design iteration of the DSVL and environment to evaluate the tradeoffs made and inform the next iteration. The second was a pair of end-user evaluations using a combination of tool familiarization and a survey instrument.

The CD evaluation at the final iteration gave the following results. The underlying "inner join" (*Choose* statements) based nature of report specification requirements naturally leads to *hard mental operations*, particularly for the non-programmer. We mitigated these by using layout (including automatic layout) in a first class way (rather than as *secondary notation*), annotation mechanisms to emphasize both the sequential and hierarchical nature of some aspects of the specification and context sensitive helpers. This approach emphasizes the report *logic* over the final report *layout* (more conventional report designers emphasize the latter) which is a tradeoff of concreteness of the final report versus a better *closeness of mapping* to the information access requirements for the data constituting the report. The latter was chosen to emphasize as Prism felt this was the area where most report specification errors occurred. The auto layout mechanisms also assist in reducing *viscosity*. Simple refactoring support, such as variable renaming, also reduces *viscosity*.

The DSVL uses a fairly *terse* set of simple color coded shapes with iconic annotations (e.g. a printer icon) as its basic notation. These are sufficient to cover the high level constructs needed, with a more *verbose* set of textual elaborations for lower level functions, for example, for reusable functions and meta-data links to the underlying Prism database. To mitigate the *diffuseness* and potential *error proneness* of these, features such as semantic checks that prevent elements and functions being used in inappropriate places are provided. This and the application of other syntactic and semantic constraints led to the almost complete elimination of spelling errors and other minor syntactic mistakes also reducing *error proneness*. Features, such as elision, support high level *visibility* and DSVL code *juxtaposition* at the expense of creating *hidden dependencies* to the elided code. This tradeoff was considered worthwhile to allow users to better obtain an overview of the report structure. Mechanisms such as co-selection of elements where other dependencies arose (e.g. specification/use of variables) served to reduce other types of *hidden dependencies*.

The two user evaluations targeted different end-user bases: 1) Prism RWL developers, i.e. those expected to maintain and extend the RWL meta-model (code base) and 2) End-users, i.e. those people regularly writing report specifications. For the developers, our focus was on the understandability and extensibility of the meta-model and associated visual notation, while for report authors our focus was on the usability of the prototype visual report specification

environment. Six developers and five end-users, all Prism employees, took part in these evaluations. The relatively small numbers meant qualitative information was sought. In both cases a brief tutorial was provided and the users then performed specified tasks and completed a survey.

Prism developers were asked to make a minor and a comparatively major modification to the meta-model using the newly developed meta-model. They were requested to record their steps and also respond to some close ended questions which were then analyzed to determine the expressiveness and usability of the meta-model.

The end-users were asked to design two reports using the newly developed visual language using the RWL modeling tool. They were also requested to record their steps and respond to some close ended questions which were then analyzed to determine the usability of the visual notation and the tool.

Developers were very positive about the meta-model used to capture the range of RWL concepts. They reported it made understanding and changing the meta-model much easier even for those who had little experience with the RWL semantics. Changing the meta-model and performing two extension tasks (part of our developer survey) also helped developers understand the meta-model and how the various constructs were related better. The meta-model helped end-users avoid making mistakes and also made it easier for the developer to define constraints as opposed to defining constraints in an arbitrary text based form. Developers found that linking visual elements and properties to the meta-model provides a consistent way to ensure semantically correct report models were created. Our model-driven approach [*20*] was seen as a good basis for generating other code if need be; such as configuration scripts.

However developers did perceive that meta-model scalability and navigation would be problematic as the model grew in size. There is a large learning curve involved with Microsoft DSL Tools and inexperience with model-driven development meant that the fundamental Model Driven Engineering (MDE) concept of generating code via transforming templates was not always intuitive to Prism developers.

Target report author end-users were particularly positive about the support the visual report writer language and tool provided for validation while designing Prism reports; they saw this as greatly saving time and effort. They found the Prism database browser for meta-data linking helped determine the current field or table to use accurately. The tool prevented report authors from making spelling mistakes and allowed them to obtain a clear overview of the database and its tables without understanding intimate details of its structure. Visually designing a Prism report, based on existing Prism RWL concepts, allowed report authors to clearly visualize the report script and its logical flow. Participants noted that other report tools, whether focused on concrete or abstract end report layout, do not provide as good a control flow and element relationship visualization as our prototype. Large report changes are well-supported by the visual tool compared to textual RWL and other report tools.

However report authors found making trivial changes difficult using the visual notation. A small change such as adding a space in a line specification requires a number of steps as opposed to doing it textually. This confirms high viscosity at low levels of detail. There is also no synchronized view between the visual notation and the generated textual RWL script-concreteness. This results in poor *juxtaposibility*. Many end-users found the Visual Studio Shell *(Experimental Hive)* hosting the DSVL difficult to use. The visual notation was felt to be primarily suitable for novice and intermediate report designers as it does not allow a high degree of customization when compared to textually designing a report. Due to this reason expert report designers felt they would be constrained by the features of the tool and this may hamper their productivity.

Overall the Prism report developers found the model-driven approach to designing and building the visual report designer prototype very promising. Prism plans to further explore this approach for a deliverable visual report designer and to inform further development of the textual report scripting language and engine.

Novice and intermediate report designers found the visual reporting language and prototype support tool effective for increasing productivity and accuracy of report writing. Expert report users are not so well supported by the visual tool in its current form.

Future enhancements include adding incremental report generation and running, showing report designers the intermediate results of their evolving report design via *progressive evaluation*. A complementary WYSIWYG view showing report bands, literals and database content more closely matching the final report would not replace our existing visual notation but augment it. A visual debugger using the visual report authoring language to step through a running report would similarly aid complex report authoring and debugging.

7. Summary

We have developed a prototype visual report writing language and support tool using Microsoft Visual Studio DSL Tools for a commercial company in the print industry. This tool complements the existing

proprietary textual reporting language and engine for authoring custom enterprise system reports. Features of our approach include the use of model-driven development via meta-model, constraints, visual notation and code generators to realize the prototype visual language tool and enable much easier meta-model and visual language extension. Evaluations of both the model-driven engineering approach for visual languages and our prototype visual report designer language and tool suggest both are promising.

8. References

[1] P. Panos and R. Weaver, "Factors Affecting the Acceptance of a Report Writer Software Application Within Two Social Service Agencies," *Journal of Technology in Human Services*, vol. 19(4), 2002.

[2] D. Atkins, T. Ball, G. Burns, and K. Cox, "Mawl: a domain-specific language for form-based services," in *IEEE Transactions on Software Engineering*, vol. 25 (3), May/June 1999.

[3] C. Pountney, *The Visual FoxPro Report Writer*. Hentzenwerke, 2002.

[4] C. Weaver, "Coordinated Queries: A Domain Specific Language for Exploratory Development of Multiview Visualizations," in *IEEE Symposium on Visual Languages and Human-centric Computing*, Herrsching am Ammersee, Germany, September 2008.

[5] R. Abraham, "FoXQ - XQuery by Forms," in *2003 IEEE Symposia on Human Centric Computing Languages and Environments*, Auckland, New Zealand, October 2003.

[6] Taguchi, Mitsuhisa, and T. Tokuda, "A Visual Approach for Generating Server Page Type Web Applications Based on Template Method," in *IEEE Symposium on Visual/Multimedia Software Engineering*, Auckland, October 2003.

[7] Microsoft. (2007) Microsoft. [Online]. http://msdn.microsoft.com/en-us/library/bb126327(vs.80,printer).aspx

[8] Prism Group. (2008) Prism - Better information. Better business. [Online]. http://www.prism-world.com/

[9] Prism Group, *Prism WIN - Report Writer Handbook*. Auckland, 2005.

[10] M. Gunderloy. (2004, Jan.) A review of Crystal Reports V10 Advanced Developer Edition and MetaEdit+ 4.0, Application Development Tools. Internet.

[11] Microsoft. (2008) Defining a Report Layout (Visual Studio Report Designer). World Wide Web.

[12] Fast Reports Inc. (2009, Apr.) FastReport Studio. [Online]. http://fast-report.com

[13] Perpetuum Software. Report Sharp Shooter. [Online]. http://www.perpetuumsoft.com

[14] Papantonakis, Anthony, and P. J. H. King, "Syntax and Semantics of Gql, a Graphical Query Language," *Journal of Visual Languages and Computing*, vol. Special Issue on Visual Query Systems, Mar. 1995.

[15] S. Di Martino, et al., "A WebML-based Visual Language for the Development of Web GIS Applications," in *IEEE Symposium on Visual Languages and Human-centric Computing*, Coeur d'Alene, Idaho, USA, 23-27 September 2007.

[16] D. Dotan and R. Pinter, "HyperFlow: an Integrated Visual Query and Dataflow Language for End-User Information Analysis," in *VLHCC05*, Dallas, Texas, September 2005.

[17] M. C. Humphrey, "A graphical notation for the design of information visualizations," *International Journal of Human Computer Studies*, vol. 50, no. 2, pp. 145-192, 1999.

[18] A. Leff and J. Rayfield, "Relational Blocks: A Visual Dataflow Language for Relational Web-Applications," in *IEEE Symposium on Visual Languages and Human-centric Computing*, Coeur d'Alene, Idaho, USA, 23-27 September 2007.

[19] M. Afonson, R. Vogel, and J. Teixeira, "From Code Centric to Model Centric Software Engineering: Practical case study of MDD infusion in a Systems Integration Company," in *Proceedings of the Fourth Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for Pervaisve and Embedded Software*, 2006.

[20] S. Beydeda, M. Book, and V. Gruhn, *Model-Driven Software Development,* . Springer, 2005.

[21] S. Cook, G. Jones, S. Ken, and A. C. Wills, *Domain-Specific Development with Visual Studio DSL Tools*. Boston: Addison-Wesley, 2007.

[22] MetaCase. (2008) MetaEdit+ Modeler - Supports your modeling language. [Online]. http://www.metacase.com/mep/

[23] T. R. G. Green and M. Petre, "Usability analysis of visual programming environments: a 'cognitive dimensions' framework," *Journal of Visual Languages and Computing*, vol. 7, pp. 131-174, 1997.

[24] J. C. Grundy, J. G. Hosking, K. Liu, and J. Huh, "Marama: an Eclipse meta-toolset for generating multi-view environments," in *ICSE08*, Leipzig, May 2008.