Augmenting DSVL Meta-Tools with Pattern Specification, Instantiation and Reuse

Karen Li[1], John Hosking[1], John Grundy[2], Tony Ly[1] and Brian Webb[1]

[1] {k.li, j.hosking}@auckland.ac.nz, {triadle, webb.brian}@gmail.com
Departments of Computer Science, University of Auckland,
Private Bag 92019, Auckland, New Zealand

[2] jgrundy@swin.edu.au
[2]Faculty of Information and Communication Technologies, Swinburne University of Technology, PO Box 218, Hawthorn, Victoria, Australia

Abstract: This paper describes an approach for using patterns in domain-specific visual language (DSVL) meta-tools. Our approach facilitates DSVL development via high level design-for-reuse and design-by-reuse pattern modelling tools. It provides a simple visual pattern modelling language that is used in parallel with DSVL meta-model specifications for modelling and reusing DSVL structural and behavioural design patterns. It also provides tool support for instantiating and visualising structural patterns, as well as executing behavioural patterns on DSVL model instances.

Keywords: Meta-tools, domain-specific visual languages, design patterns, code generation, model-driven engineering

1 Introduction

Using DSVLs (a.k.a. DSMLs or DSLs) for software development has gained more awareness in industries, open source communities and software tool vendors nowadays, thanks to their offering of high level visual representations of domain-specific knowledge, making it possible for stakeholders to be more deeply involved in software development. We have previously described our Eclipse-based Marama meta-toolset for generating multi-view DSVL environments in [GHHL08]. Marama features rapid specification of meta-models, visual notations, views, constraints, critics, event handlers, model transformations and code generations for DSVLs. However, its support for reuse is currently very limited. Many other meta-tools (e.g. [Mic08, KLR96, LBM+01, ZGH+07]) have attempted to support design-by-reuse, however, most approaches are still limited to code-level (e.g. via white-box framework inheritance or composition) or fine-grained modular model-level reuse (e.g. via copy/paste and import/export). Only few of them (e.g. [Sut02]) have addressed larger trunk or higher level reuse with emphasis on patterns.

Through our research with various visual languages and tools, we have identified a series of recurring problems and solutions for DSVL design and implementation. These include patterns for the specification of model structures such as hierarchy, composability, cardinality,

mutability, multiple linked views and model interoperability; and patterns for the specification of modelling behaviours for common visual analytics tasks such as retrieving model data of interest to create visualisations, detecting and removing conflicts, transforming views and visualisations, and diffing and merging models/views. In addition, we have discovered a common set of repeatedly used relationships including sub-typing, containment, referencing, dependency, flow, mapping and merging. We are investigating these DSVL design patterns and their internal and external relationships in terms of their generic specification (via a high-level visual language), instantiation/execution and reuse (adoption, adaptation, composition and inheritance).

This paper describes our contribution about a generic but configurable meta-model level visual language and tool support for DSVL design pattern (both structural and behavioural) specification, application and reuse. We begin by describing the motivation, related work and the overall tool architecture of our approach highlighting how loose-coupled structural and behavioural pattern specifications are integrated coherently with our consistent reuse and configuration support. We then separately describe structural pattern specification and instantiation followed by behavioural pattern specification and execution in MaramaDSL. We then discuss early analysis results and future research before we conclude the paper.

2 Our Approach

Our initial motivation for this research came from one of the lessons we learned from our intensive evaluations [Li07] on Marama, which is that although we have used simple metaphoric visual languages (multiple integrated DSVLs for specifying structural and behavioural aspects) that map well to the problem domain for DSVL design specifications, end users are still confronted with a steep learning curve and hard mental operations when dealing with complex designs that have both structural and behavioural DSVL aspects. This raises a major barrier to use. However, we have identified that a good number of design specifications are just generic ones (or can be generalised) that can be reused across domains. We believe that facilitating design-for-reuse and design-by-reuse via patterns [Sut02] is an optimal way for removing this barrier.

We aim to capture common aspects of design and implementation support for different DSVLs, and facilitate pattern-based reuse to augment DSVL meta-tools. We want a family of modelling notations and tools for DSVL pattern specification, visualisation, instantiation and execution. We also want to allow easy reuse of DSVL patterns via language and environment support.

Our earlier work on abstract design pattern specification, DPML, has been reported in [MHG07]. DPML defined a visual design pattern modelling notation and provided tool support for pattern instantiation on a UML object model. DPML provides relatively clean visual representations for various types of pattern participants (e.g. interfaces, methods and operations), their dimensions (collections) and constraints. The instantiation step permits both tailoring and traceability/consistency with pattern specifications. However, DPML's meta-model was based on the UML meta-model, which constrains its adaptation to a wider range of DSVL application domains (such as the performance engineering, business modelling and health care planning illustrated in [GHHL08]) and its integration in a generic DSVL meta-tool

for design and implementation of non-UML-based visual language notations for those domains. Other existing UML-based pattern languages (e.g. [FKGS04, MCL04]) are likewise. Although they have enough expressive power in specifying traditional design patterns, they are not flexible in collaboration with arbitrary DSVLs.

Graph grammar based transformation techniques have also been used to specify design patterns [ZKDZ07]. However, the nature of graph transformation (pattern matching through a left-hand side rule and transformation through left to right-hand side rule mappings) makes it more suitable to define model evolution/refactoring changes instead of up-front pattern specification and instantiation. Configurability also needs to be extended in order to allow reuse of generic rules.

Our current approach targets to facilitate simple and holistic pattern support in DSVL meta-tools for the design and construction of arbitrary DSVLs and their supporting environments. To this end, we have designed a notation that explicitly models generic pattern participant roles and relationships, with the ability to accommodate variance of different DSVL meta-models through visual, meta-level configurations. Our approach allows both domain specific patterns (such as the "Abstract Factory" pattern specified and instantiated on UML design models) and generic design patterns that can be reused across multiple DSVL domains. Our visual notation has been designed based on a theoretical foundation [Moo09] so as to make it cognitively manageable by non-programmer DSVL end users.

We have designed MaramaDSL, a tightly integrated meta-tool environment featuring quick and easy to use pattern-oriented modelling tools. It was created using Microsoft DSL Tools [Mic08], deployed to the Visual Studio IDE and used back to augment the DSL Tools with extended designer views, framework code and code generators. Figure 1 illustrates its usage architecture. A MaramaDSL editor (with multiple designer views) is added into a DSL project to co-function with the DSL Designer (Figure 1 (a) – DSL Tools' visual definer for domain classes, relationships, shapes, connectors and diagram element maps). The complementary MaramaDSL model (with user created pattern-related components) generates custom code onto the DSL project so it realises in the generated DSVL environment without the need for any additional configuration. MaramaDSL imports the DSL Designer elements and provides a Domain Model View (Figure 1 (b)) which is used to display the existing DSVL meta-model elements, but in a flattened (without trees) way with filtering choices for various components (e.g. relationship role links) in order to manage diagram cleanness. Two pattern specification views linked with the Domain Model View are available. They are the Structural Pattern Designer View for structural pattern specifications (Figure 1 (c)) and the Behavioural Pattern Designer View for behavioural pattern specifications (Figure 1 (d)), each of which exploits parallel orthogonal representations for separate but easy to bind generic pattern specifications and contexts (with shared use of the DSVL meta-model in a layer as the pattern specification context, providing additional filtering capabilities for adding in interested potential domain elements for pattern participation).

Patterns are specified in two simple visual languages (comprising different structural and behavioural specification notations and metaphors) that are described in the next sections. With regard to the environmental support for design-for-reuse and design-by-reuse, specified

patterns can be saved context-free (with all context bindings removed), appearing in a Patterns Explorer Tree in the Model Explorer window; they can then be explored and drag-dropped from there for direct reuse and binding with other DSVL meta-models. Accessed pattern specifications can also be easily adapted for reuse in a variant way (introducing variability based on the commonality), e.g. modify or remove any existing participant or relationship at the DSVL client, or add elements to a pattern specification to meet specific needs. MaramaDSL also allows complex patterns to be created by composing existing patterns (as sub patterns). We will demonstrate the appropriateness of such pattern composition in Section 4 using a composite behavioural pattern example. We are currently expanding a set of generic DSVL patterns for simple reuse and composition purposes.

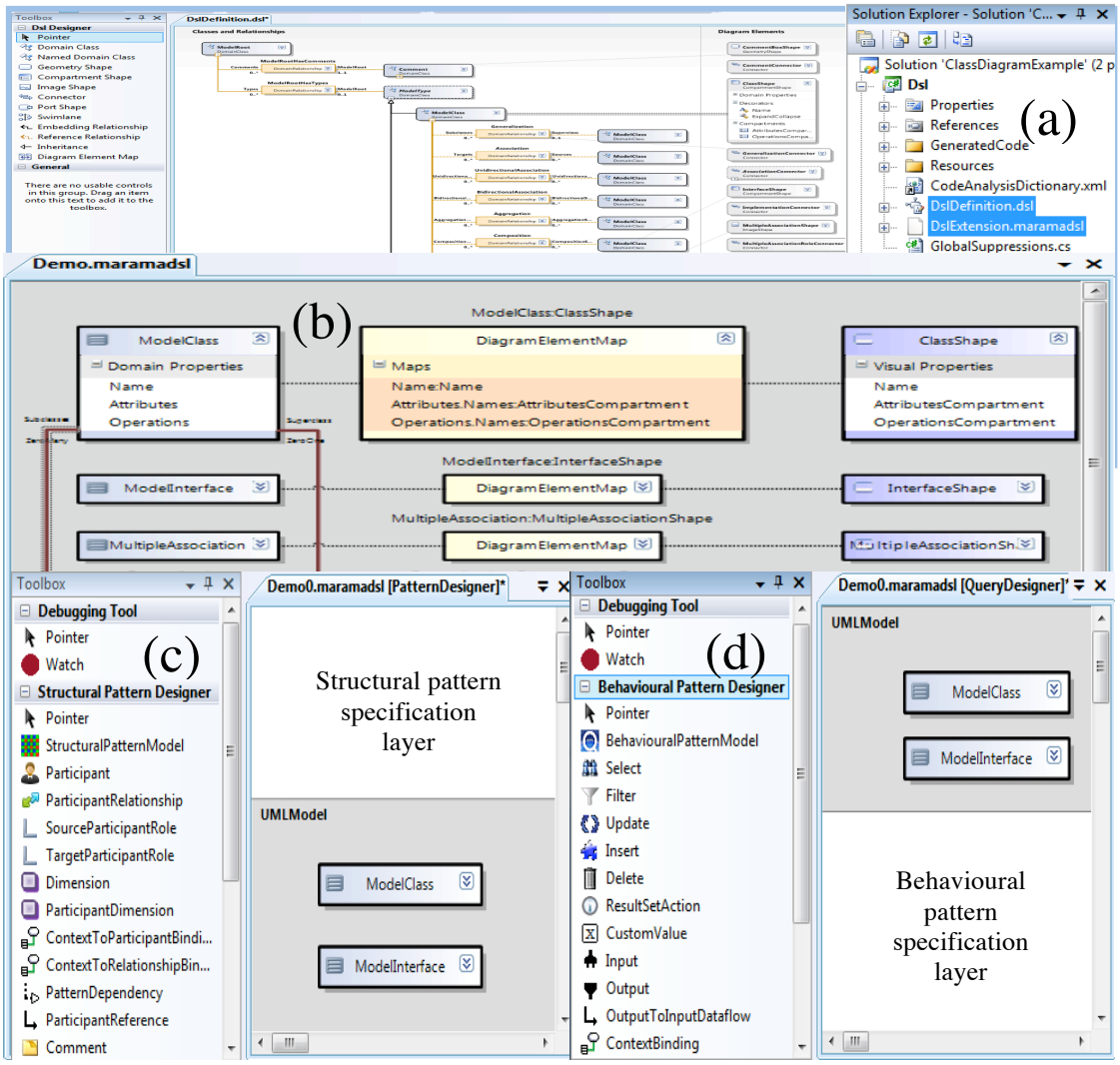


Figure 1. MaramaDSL usage architecture: (a) Integration with DSL Designer, (b) Domain Model View, (c) Structural Pattern Designer View and (d) Behavioural Pattern Designer View

We exploited a set of consistent modelling and visualisation techniques in the design of MaramaDSL, aiming to provide end users with a consistent and cognitively manageable user experience. Different visual language metaphors are used for structural and behavioural pattern specifications to accommodate end users' mind maps (to be elaborated in the next sections). Drag-drops are largely used to enable creation and reuse, while links are used for bindings of pattern roles and contexts. An overlay layer of annotations are used to expose constraints and dimensions. Orthogonal layers of pattern modelling elements and the DSVL meta-model are juxtaposed for separated visualisation with cross-cutting concerns and convenient specification. Multiple interacting views (domain model, structural and behavioural pattern perspectives) are also used together to complement one another within a unified underlying model. Complex specifications (both single and composite elements) can be collapsed and filtered, and relationships and links can be concealed on demand in order to manage diagram clutter.

3 Structural Pattern Specification and Instantiation

Our structural pattern specification language uses an entity-relationship based metaphor and uses a visual notation as shown in Figure 2 for depicting (a) patterns – represented by container shapes with a pattern icon and a name text decorator; (b) participants – represented by inner shapes with a participant icon and text decorators for name, type and any bound domain context; (c) participant relationships - represented by inner shapes with a connection icon and text decorators for type and any bound domain context; (d) dimensions – represented by inner shapes with a dimension icon and a name text decorator, as well as by port shapes on participants (the dimension concept is based on DPML [MHG07], which complements the participant relationship with explicit multi-dimensional participant role cardinality constraints); (e) constraints - represented by port shapes on pattern elements to constrain participants and relationships. Constraint specifications are currently set as C# expressions (for example, a constraint imposed on a pattern may contain an expression "participant1!=participant2" denoting that two participants can not be the same runtime instance), but we intend to replace this with OCL features in a Spreadsheet metaphor similar to MaramaTatau in [Li07].

A set of pattern categories, participant and relationship types have been defined for selection based on our research and experience in the DSVL problem domain. Examples include metrics, multiple view, model integration, query and process pattern categories; domain class, property and relationship, shape and connector participant types; sub-typing, containment, and referencing participant relationship types; and using, refinement, and dependency pattern relationship types. Custom categories and types can also be added as a form of end user extensions.

The binding of a structural pattern specification to a DSVL meta-model is visually supported via static context bindings (Figure 2 (f) – represented by green dotted lines connecting elements in the domain model with their specifications in the pattern specification layer). These are cross-cutting relationships between elements from the two layers (DSVL meta-model layer and structural pattern specification layer), e.g. a domain class contextualises a participant; and a domain relationship contextualises a participant relationship. The context binding links can be concealed at individual pattern element level for diagram clutter

management. Context bindings are supplemented by a dual text encoding on a pattern element (via underlined text in the bound pattern element) to ease context navigations. A specified pattern can be saved context-free (with all context bindings removed), and will appear in the Patterns Explorer Tree (Figure 2 (g)), for reuse into other DSVL meta-models (by drag-dropping the pattern from the Patterns Explorer Tree to a Pattern Designer View followed by context bindings).

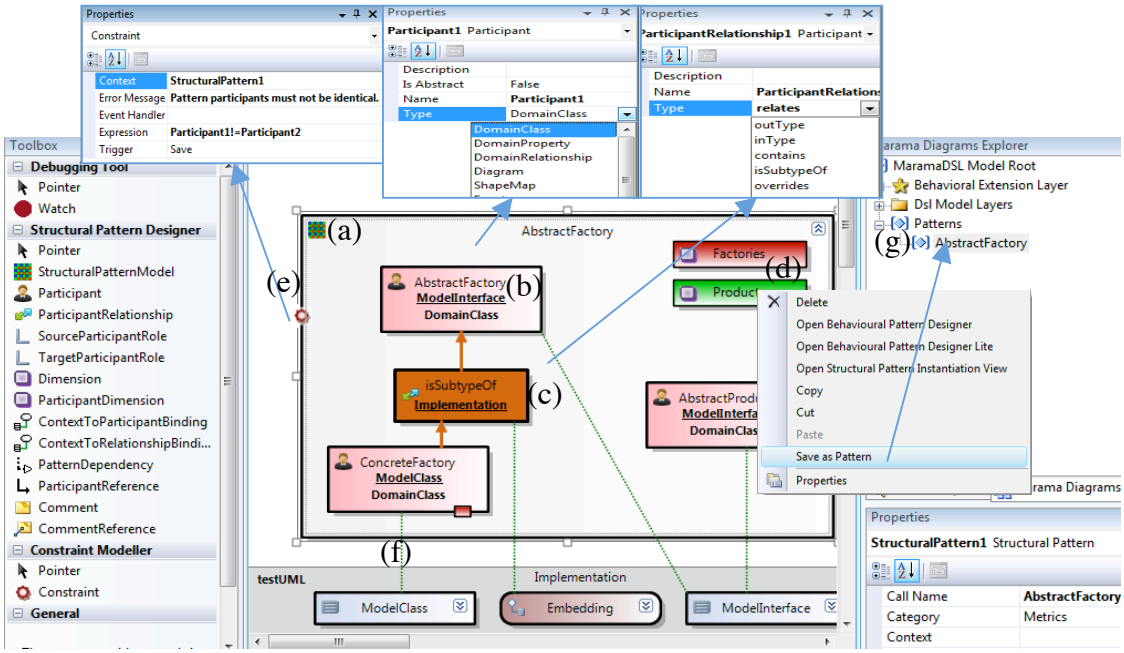


Figure 2. Structural pattern specification in MaramaDSL

After static bindings have been defined, the DSVL meta-model on the DSL Designer side is injected with pattern attributes, e.g. patterns and participant names, which are further specified by dynamic bindings on a DSVL model instance (this process is called pattern instantiation [MHG07], which creates a pattern instance associating a DSVL model). A Structural Pattern Instantiation View (Figure 3 (b) based on and linked to the Structural Pattern Designer View design-level abstraction) is provided for such dynamic bindings. Participant memberships can be added into the pattern instance specification. On such pattern instantiation, either completed or not, the DSVL model instance (Figure 3 (c)) is updated with either creation (e.g. creating model elements and relationships if they didn't exist before pattern instantiation) or modification (e.g. resolving relationship conflicts by removing/changing existing relationships) through an extended generic validation procedure (future implementation). These are the generative effects of pattern application, facilitating both up-front design as well as design refactoring. Multiple patterns can be specified for a DSVL meta-model and instantiated onto its model instances. Traceability of pattern role bindings on the DSVL model instance is supported via a "Pattern Message Board" (using a "dashboard" metaphor), which allows interactive selection highlights of pattern participants, i.e. selecting a participant in the "Pattern Message Board" will highlight all the participant members in the DSVL model instance.

We revisit the Abstract Factory pattern example specified in DPML in [MHG07] and illustrate its new "skinning" in MaramaDSL. By repeating this example, we emphasise the generalness as well as the configurability of our new language. Figure 3 (a) shows the pattern specification in parallel with the UML meta-model. The pattern contains six participants (Abstract Factory, Concrete Factory, Abstract Creator, Concrete Creator, Abstract Product and Concrete Product) and their internal relationships ("isSubtypeOf", "contains", "overrides", and "outType"). Context type bindings include that a UML Model Interface will participate as an Abstract Factory; a UML Model Class will participate as a Concrete Factory, a UML Implementation relationship will be used to link the Abstract Factory and the Concrete Factory, and so on. Two dimensions are defined in the pattern: Products and Factories. They are selected by various participants as their coloured overlay annotations indicate, and they are used to specify that the number of participant instance members (as shown later in Figure 3 (b)) should be subject to the numbers of dynamic items added to the dimensions (with cross product of the numbers if more than one dimension is set on a participant [MHG07]). Figure 3 (b) shows the Structural Pattern Instantiation View associated with a specific UML model instance, in this case showing the actual UML elements that are participants in the various roles in the Abstract Factory pattern. Pattern creation effects, including creation of participants and relationships of the right types on the DSVL model instance, are established synchronously based on the pattern instantiation. For instance, on adding a MetalFactory member for the Concrete Factory participant in the Structural Pattern Instantiation View, a UML Model Class is created with that name on the DSVL model instance and an Implementation relationship is created linking it to the Abstract Factory member – GUIFactory, as shown in Figure 3 (c).

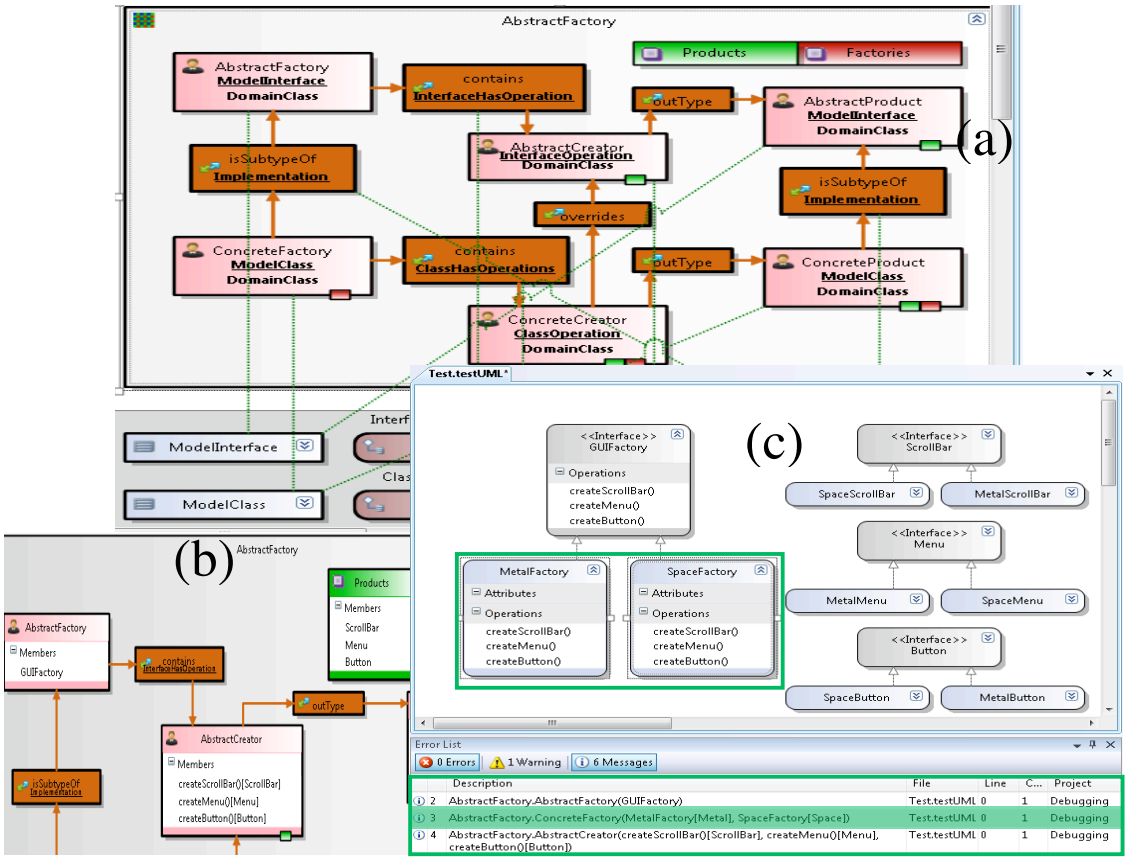


Figure 3. Abstract Factory pattern specification and instantiation on a UML model

4 Behavioural Pattern Specification and Execution

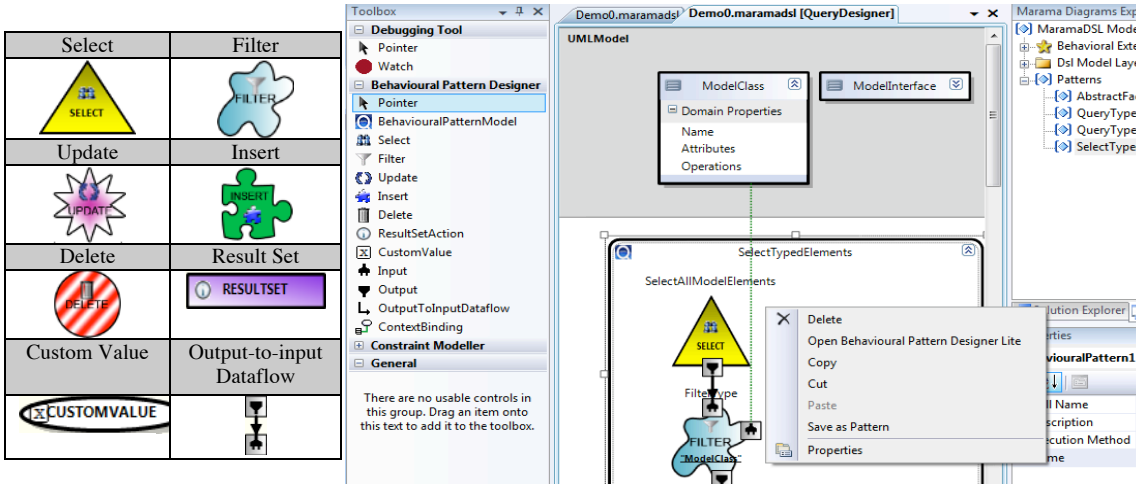
We initially tried to use a uniform entity-relationship (ER) based visual language representation for both structural and behavioural pattern specifications, allowing behaviours to be represented as participants of Event, Query, Filter, or Action types. However, the ER metaphor proved a failure for these purposes. We instead used a more traditional dataflow-based metaphor for behavioural pattern specifications and represent behaviours as executable queries and actions which are composed by a set of generalised query and action elements.

To develop this approach, we initially implemented by hand 40+ behavioural specification examples on different domain models, performing visual analytics tasks such as content retrieval, conflict management and information aggregation. We examined the resulting manual codings which were largely duplicated and identified common abstractions, from which we have obtained a vocabulary for behavioural specifications. The vocabulary includes a set (50+) of query building blocks specialising the following query and action types: SELECT, FILTER, UPDATE, INSERT and DELETE, plus a set for RESULTSET rendering (resulting from actions on the elements selected by the query). They are used to retrieve data, set filtering criteria, or alter model/view elements for various query-based visual analytics tasks. The elements are all parameterized with query context (e.g. model/view or

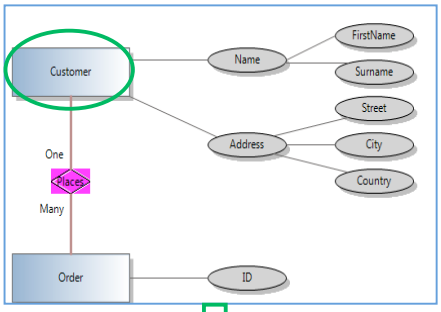
element/relationship/ shape/connector parts) and criteria (e.g. typed value). Each has a returning result state as the output. We have made them self-descriptive and type sensitive so as to mitigate ambiguity in use. Our developers have found behavioural composition using these elements to be much easier than the original ER metaphor we tried, requiring then to just place necessary building blocks and pipe the output from one to input of another. We wanted to allow non-programmer DSVL end users to take advantage of the generalised query and action elements to create behaviours via a visual language that represents the elements visually, and defines how they interact with each other to facilitate behaviour composition. We also wanted to minimise behavioural specification effort by providing high-level behavioural pattern reuse.

In this behavioural pattern specification language, we provide a Behavioural Pattern Specification View with graphical symbols with in- and out-ports and links between to express what should happen (i.e. state retrieval, state modification and output generation) after a given data push or pull. The symbols include Behavioural Pattern Model, Select, Filter, Update, Insert, Delete, Result Set, Custom Value, and Port Dataflow in the visual forms shown in the table in Figure 4. We used different shapes to represent queries, actions, result sets and values, and different icons, colours and textures for different query and action types. Horizontal and vertical positions are used at a Behavioural Pattern Model level to express dataflow sequences in a natural way. For each visual symbol, text also plays an important role in determining the actual calling building block. We integrate a DSVL meta-model with behavioural specification to allow convenient context bindings, using the consistent and shared DSVL meta-model layer, context binding links and dual text encodings of bound elements as in the Structural Pattern Designer View (as described in Section 3), apart from the presentation that the DSVL meta-model layer is shifted to the top in this view to better fit the top-down data piping style in behaviour composition. Every Behavioural Pattern model has two execution methods that the user can specify: automatic or controlled. The former enables automatic execution based on user-diagram interaction, the latter provides context menus to allow users to click and run the behaviours. A consistent reuse mechanism as for the structural patterns is available for behavioural pattern specifications.

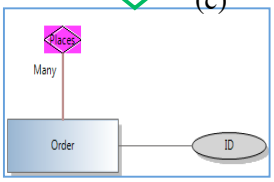
(a)

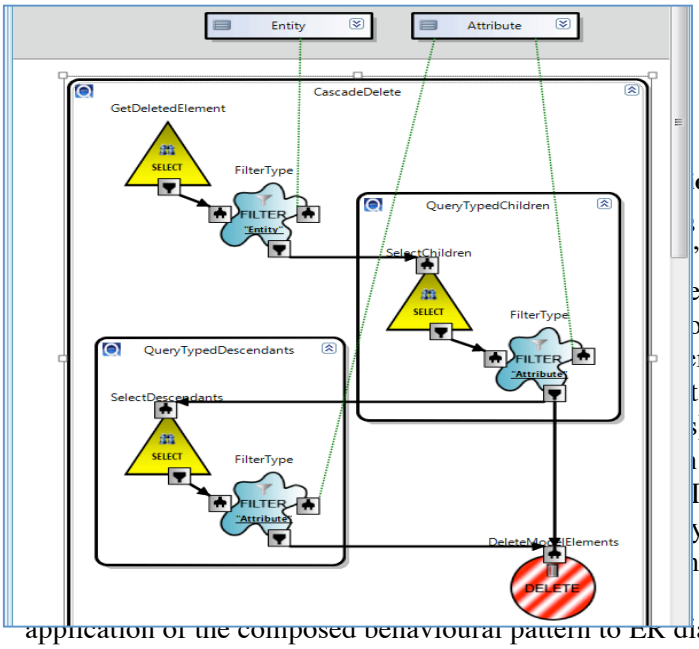


(b)



(c)





on in MaramaDSL

a Behavioural Pattern Model which
' operation where the filter is bound in
ecting all ModelClass elements in the
own in Figure 4 (b). Here a composed
rform a cascade delete behaviour. It
ts associated attributes as well as their
s) should all be deleted, as shown in
composes two earlier defined generic
TypedDecendants, which express the
ypes. It also contains additional query
posed behavioural pattern model as
bindings to the domain model for
application of the composed behavioural pattern to ER diagrams.

5 Analysis

Our approach explicitly models pattern participant roles and relationships for easy configuration across DSVLs. Compared to UML-based approaches (e.g. [FKGS04, MCL04, MHG07]), it has fewer constructs (e.g. no inheritance, aggregation, interface or operation) and thus simpler; it has a clearer role collaboration model (participants, relationships, dimensions, constraints and behaviours) specific to the pattern description domain thus is more visually expressive for pattern specific concepts; it allows easier domain context binding of generic pattern models through decoupled but interacting parts; and it allows sharing of common pattern structure for better reuse of specification (e.g. the State and Strategy patterns have a common structure, and by using our approach one specification can be easily adapted based on the whole other without the need to re-specify common individual parts).

Our solution provides appropriate abstractions, simple-to-use visual notations and high-level instantiation, execution and reuse support for both structural and behavioural pattern functionalities to be realised in DSVL tools. We have applied the Physics of Notations theory [Moo09] for a principled visual language design:

- Our language supports *Cognitive Integration* by using multiple linked views with separated layer representations of sharable and filterable domain context elements; it also provides consistent design-for-reuse and design-by-reuse mechanisms for structural and behavioural pattern specifications and context bindings.
- *Graphic Economy* is the dominant principle in our language, for which some tradeoffs were made. We chose to use "symbol overload" (e.g. one participant symbol representing various types of pattern participants; one query symbol representing multiple behavioural building blocks) in both the structural and behavioural pattern specifications. This resulted in negative *Semiotic Clarity*, but this is however desirable to limit diagram and graphical complexity with a cognitively manageable number of symbols. As a result, our language heavily relies on text *Dual Coding* to distinguish pattern elements.
- We applied the *Principle of Perceptual Discriminability* using colours and iconic annotations as the primary means of visual distance for distinguishing pattern elements from each other, though we didn't use the full range of visual variables (horizontal and vertical position, size, brightness, colour, texture, shape and orientation) urged in the *Principle of Visual Expressiveness*, the multiple channels used in the current form have sufficient visual expressiveness. The *Principle of Semantic Transparency* is addressed by using depicting icons to suggest symbol semantics.
- Some visual patterns can become complex with many symbols. Applying the *Principle of Complexity Management*, we introduced a mechanism that collapses selected pattern elements into a sub pattern using the same container symbol but in a collapsed form. The sub pattern can be expanded for observation and modification. Form-based filtering show/hide (to turn selected elements and links on and off) features are also supported. They are useful for flexible information display and manipulation. The mechanism to reuse existing patterns also assists in managing complexity. For behavioural pattern specifications, we have also provided a Behavioural Pattern Designer Lite View (not illustrated in the paper due to limit space) (with elided technical details such as ports and context bindings, using mainly descriptions and flows) to address the *Principle of Cognitive Fit* by exhibiting multiple dialects. The Behavioural Pattern Designer Lite View essentially represents the same concept of the Behavioural Pattern Designer View, can perform the same tasks, and translate to the same underlying query and action elements. It benefits users wanting high level specifications while the Behavioural Pattern Designer View is for users needing behavioural composition details. The ability to convert between the forms provides users useful alternative perspectives.

We have also conducted a Cognitive Dimensions [GP96] analysis to evaluate tradeoffs, strengths and weaknesses of our solution. The visual language has expressiveness equivalent to domain-specific code written with APIs, but with a lower *abstraction gradient*, augmented understanding, reduced effort, and a much shallower learning curve via better *closeness of mapping* to users' mental models of pattern use. Instantiating a pattern specification requires some *hard mental operations* and *premature commitment* when choosing appropriate pattern elements to compose. However adding *abstractions* in the form of pre-defined patterns reduces

complexity and *diffuseness*. The use of the visual language reduces *error proneness* compared to coding, but requires proactive checking of model semantics for correctness. *Progressive evaluation* is allowed but requires a compile-and-run cycle for the generated code. The language uses a *terse* set of graphical symbols but with a rather *verbose* set of textual labels for expressing pattern elements. *Diffuseness* caused by that is mitigated by using them within typed symbol groups. The visual symbols have clear *role expressiveness*. We use layout in behavioural pattern specification as a *secondary notation* because it does not affect any semantics but is good for promoting readability of the flow sequences with visual cues. The usual diagram insert *viscosity* problems occur, and require automatic layout to mitigate. We have mitigated areas of *hidden dependency* and v*isibility* in the language by *juxtaposition* of orthogonal layered views, and dual coding of custom values though context links and dynamic properties.

The preliminary analyses also helped to identify missing functionalities that require future work. DSVL design patterns can offer some level of quality criteria to domain-specific modelling, as validated design patterns are quality design models themselves, and so correctly applying them onto DSVLs means imposing a quality shield on them. However, we are yet to address how we can validate the DSVL design patterns and their instantiations/executions for completeness, consistency and soundness. Another important issue raised during the analyses is that conflicts do exist in design patterns and when a pattern language is allowed to be used, balancing the conflicts and providing users with decision support for pattern adoption becomes essential. To address this, we plan to use Mussbacher et al's goal-oriented approach [MWA06] for forces analysis in the pattern language. Runtime animated visualisation of the execution of behavioural pattern elements was a suggested feature but is only partially supported to date. We are also planning extensive usability studies.

6 Conclusion

Our aim is to augment DSVL meta-tools with pattern-oriented design for easier end user experience. MaramaDSL, extending the DSL Tools, is a unified meta-modelling environment with both structural and behavioural pattern specification and usage built into the holistic DSVL meta-modelling based development process. MaramaDSL provides a general language for specifying DSVL design patterns with also tool support for pattern instantiation, execution and cross-domain reuse. Our ultimate goal is to facilitate a knowledge base (with formalised pattern representations) for sharing DSVL design knowledge to benefit the wider communities, and the work we demonstrated here leads towards that direction.

Bibliography

[FKGS04] R.B. France, D.K. Kim, S. Ghosh, E. Song. A UML-Based Pattern Specification Technique. IEEE Transactions on Software Engineering 30(3), 2004.

[GHHL08] J. Grundy, J. Hosking, J. Huh, K. Li. Marama: an Eclipse Meta-toolset for Generating Multi-view Environments, in ICSE'08. 2008: Leipzig, Germany.

[GP96] T.R.G. Green, M. Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. JVLC, 7: 131-174, 1996.

[KLR96] S. Kelly, K. Lyytinen, and M. Rossi. Meta Edit+: A Fully configurable Multi-User and Multi-Tool CASE Environment, in Proc. of CAiSE'96. 1996.

[LBM+01] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, G. Karsai. Composing Domain-Specific Design Environments. Computer, 2001: 44-51.

[Li07] K.N.L. Li. Visual languages for event integration specification in Computer Science. 2007, University of Auckland: Auckland.

[MCL04] J.K.H. Mak, C.S.T. Choy, and D.P.K. Lun, Precise Modeling of Design Patterns in UML, in ICSE'04. 2004: Scotland, UK.

[MHG07] D. Maplesden, J.G. Hosking, and J.C. Grundy. A Visual Language for Design Pattern Modelling and Instantiation, in Design Patterns Formalization Techniques. March 2007, Toufik Taibi (Ed), Idea Group Inc.: Hershey, USA.

[Mic08] Microsoft Domain Specific Language Tools. http://msdn.microsoft.com/en-us/vsx/default.aspx, Microsoft, 2008.

[Moo09] D.L. Moody. The "Physics" of Notations: Towards a Scientific Basis for Constructing Visual Notations in Software Engineering. IEEE TSE 2009.

[MWA06] G. Mussbacher, M. Weiss, and D. Amyot. Formalizing Architectural Patterns with the Goal-oriented Requirement Language, in VikingPlop 2006.

[Sut02] A. Sutcliffe. The domain theory: patterns for knowledge and software reuse 2002: Mahwah, N.J.: L. Erlbaum Associates.

[ZGH+07] N. Zhu, J.C. Grundy, J.G Hosking, N. Liu, S. Cao, A. Mehra. Pounamu: a meta-tool for exploratory domain-specific visual language tool development. Journal of Systems and Software, 80 (8), 2007.

[ZKDZ07] C. Zhao, J. Kong, J. Dong, K. Zhang. Pattern-based design evolution using graph transformation. JVLC 18(4), pp. 378-398, 2007.