IMPROVING MAINTENANCE OF BUSINESS POLICY CONSTRAINTS IN A DOMAIN SPECIFIC LANGUAGE

Tod Sedbrook, Monfort College of Business, University of Northern Colorado, Greeley CO 80639, 970-351-1337, tod.sedbrook@unco.edu

ABSTRACT

Developing a Domain Specific Language (DSL) for a business domain requires organization and maintenance of a complex set of business policy constraints. This study investigates a new design technique that transforms domain rules and constraints into declarative and functional forms and then applies those forms within a monadic framework. We translated constraints into F# representations and demonstrated a way to commute and compose groups of constraints into F# workflows. Composite workflows drive validation as modelers iterate with DSL tools to refine and maintain domain models.

INTRODUCTION

Business managers and system developers share responsibility for capturing requirements and designing systems that support changing sets of business processes [7]. A promising approach to improve communication among stakeholders is to capture requirements through Domain Specific Language (DSL) models [6]. A DSL is an expressive domain language designed to improve communication among business stakeholders and application developers.

The Unified Modeling Language (UML) provides formal semantic notations to model business requirements. UML's standardized notations allow designers to capture a domain's entities, processes and relationships through a series of graphical models. While UML provides a general set of building block for system designers, business domain experts are less familiar with it limiting its utility for them.

Unlike UML models, DSLs can improve communication by providing modeling tools that can be developed, understood and validated by domain experts [3]. Developing DSL tools requires meta-models (a model that creates other models) that codify a domain's policies and business rules and guides the construction of domain models. Meta-models provide customized building blocks that can be configured to express well-formed and tailored business designs.

Best practices recommend the Object Constraint Language (OCL) as the method to capture precise and unambiguous business rules within a business domain. As a formal declarative language, OCL improves the ability of business modelers to specify and validate UML domain models. For example, OCL can express business restrictions: "A customer cannot place more than three special orders" or policies: "A customer with order volume over 1000 units is eligible for a discount." The above constraints are expressed formally in OCL syntax. OCL, however, is a pure specification language and avoids implementation concerns.

Nevertheless, when OCL is used to support business designs, its meta-model representations must ultimately be translated from a text-based constraint representation into working code to assist in validating resulting designs. Code generation tools are available to translate OCL policies into procedural languages such as Java, C++ and C#. These procedural languages support an imperative style of code validation, but the declarative nature of OCL is lost during translation. That is, each

translated procedure is understandable only within context of program statements which introduces a semantic gap between business domain modelers and application developers.

Further complicating understanding of procedurally translated OCL is the need for an additional validation framework to detect exceptions and manage policy violations. Additions or changes to OCL policy constraints mean that DSL meta-modelers must, in addition to generating code translations of OCL policies, extend the validation framework to incorporate and process the updated constraints.

The following applies a design science methodology to build and evaluate a prototype [4]. We investigate the DSL problem domain through a design that addresses automatic validation of DSL models through F# validation frameworks. The F# programming language was introduced in 2010, and it is fully interoperable with other .Net language and modeling tools.

Since both OCL and F# are declarative languages, we investigate whether the translation of OCL to F# retains semantic consistency across business constraint representations. Also, we explore declarative constraint validation frameworks, called monads that may simplify additions and updates to DSL constraint sets. After applying the prototype to generate and apply OCL translated F# to support DSL business domain modelers and evaluating the effectiveness of constraint application within a DSL modeling environment, a discussion characterizes the results and explores the strength and weakness of this approach.

VALIDATING DSL MODELS

A key to creating a successful software system is developing a rich domain vocabulary for iteratively testing and combining domain policies and terms to create understandable models. A domain model provides the basis for understanding semantics and constraints that drive the development of subsequent software artifacts. Recently, the need to improve communication among domain experts and designers has led to growing research in DSLs tailored to a particular application domain [3, 5].

DSL modeling workbenches (a software system that creates DSL modeling components) provide mechanisms for creating modeling tools that allow domain modelers to arrange and validate configurations of DSL concepts. A workbench's graphical designer pane provides a drawing surface that allows manipulation of graphical shapes, representing the domain's underlying semantic concepts, and supports formation of permissible relationships among those shapes.

A DSL meta-model requires OCL to define constraints which express semantic restrictions among a domain's element and their relationships. For example, the constraint: "a pilot not certified for a specific aircraft cannot be assigned to that aircraft", represents a rule in the domain of aviation. A formal language, such as OCL, is used to codify domain constraints at the meta-modeling layer. In addition, constraints must be incorporated into executable frameworks that support both the formation and application of DSL constraints to validate concrete domain models.

The F# language is a new declarative .Net language introduced to allow designers to take advantage of functional programming paradigms. F# has attributes similar to lisp, Haskell, OCaml, ML and other functional programming languages that have demonstrated a long history of creating concise, readable and logically verifiable code. [1]. F# is fully interoperable with C++, C# and other languages that run on the Common Language Infrastructure (CLI) and leverages all existing .Net libraries [8].

Function programming defines pure functions that are easy to test and understandable since each unique input results in a single predictable output without side effects. Monads, derived from theories of computation, extend pure functions. They have proven to be useful to structure functional programs [2] and provide operations that offer a controlled way to manage functional composition by supplying generic operations for handling exceptions and reading and updating execution state [9, 10]. Monads allow groups of constraints to be applied, functionally composed and managed in a well understood and standard monadic framework. Then the framework supports constraint application, and controls success and failure conditions.

F# refers to workflows as its monadic programming paradigm. Workflows provide the prospect organizing and managing separate constraint grouping by supporting standard and extensible monadic structures.

APPLYING F# WORKFLOWS TO VALIDATE A DSL

To investigate the feasibility of applying F# workflow in managing DSL contraint networks, a research prototype was developed to extend Microsoft Visual Studio's DSL validation tools. Visual Studio supports a DSL modeling meta-language which specifies models within a meta-graphical designer, compiles those models and to produces a stand-alone graphical designer suitable for modeling a business domain. The stand-alone graphical designer model (the output of a meta-mode) allows business modelers the ability to manipulate graphical shapes to define specific configurations of domain concepts.

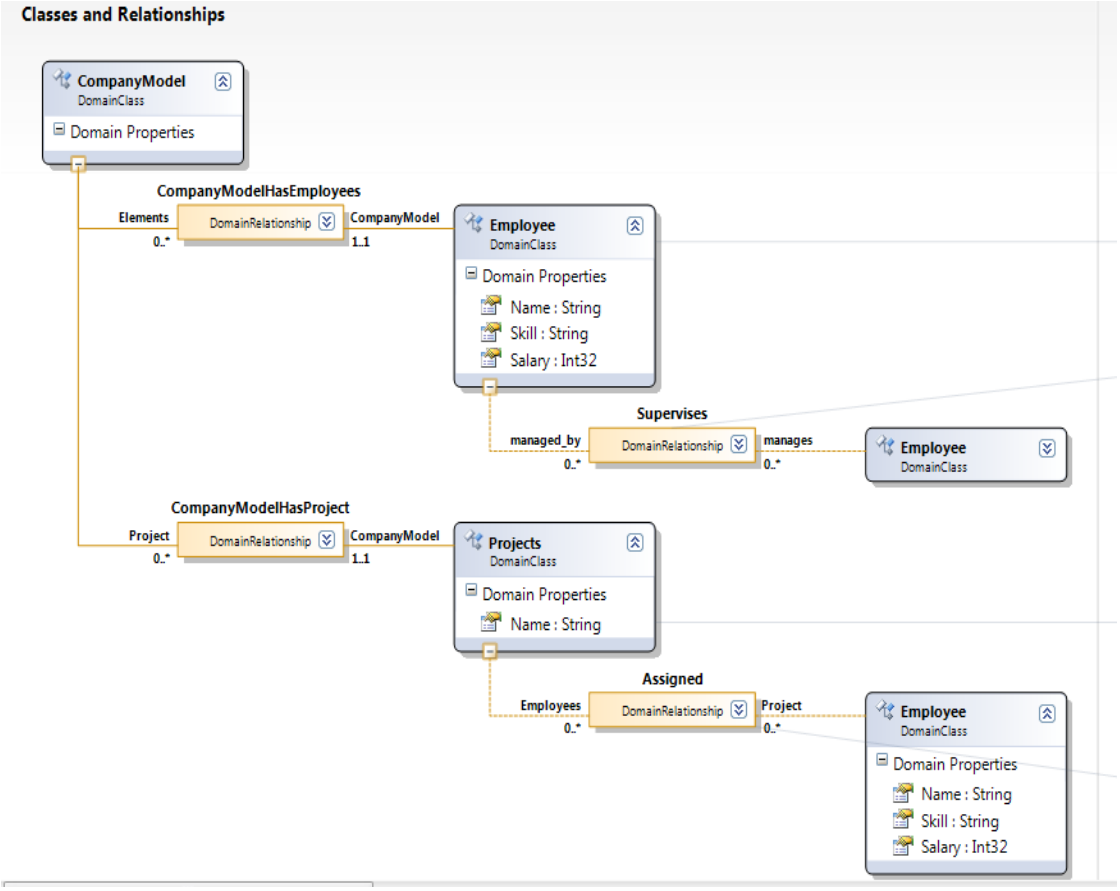
The following describes a Big Pharma, a fictitious phamcological research lab that manages several ongoing research groups working in the areas such as flu vaccines, basic biochemical research and manufacturing. Recent mergers and consolidation of company operations have forced revision of standards and policies for supervision and assignment of employees to sets of new and active laboratory projects. The company has defined project policies related to budget allocations, required technical and organizational skill sets, safety and environmental certification mixes, supervisory relationships, workloads and geographic constraints. The company follows a matrix organization where an employee might be accountable to both an assigned lab group supervisor for overall performance as well as to project supervisors for each assigned project.

Project managers represent domain experts that are charged with organizing employees into supervised project teams and then assigning employees to work on one or more projects. A DSL model is needed to assist in these two goals by allowing managers to organize an ever evolving set of policies and then comply with those policies when creating employee assignments The purpose of the DSL model is to allow project managers to explore and manipulate assignments and communicate proposed solutions to clients, managers and employees.

Figure One presents a illustrative meta-level DSL model that defines domain terms and relationship needed to define a concrete DSL models. The model defines the supervises relationship among sets of employees to define supervisors hierarchies and related groups of supervised employees.

FIGURE ONE.

Meta-level DSL model to define domain terms and relationships available to construct individual domain models for projects and employees



The DLS building block direct the formation of relationships among employees and projects and for example, prevent projects from participating in a "supervises" relationship and prevent employees form being "assigned" to other employees; however, the DLS tool's syntax does not provide a check for semantic correctness related to other policy constraints.

For example, the company has identified a salary policy that "No employee shall be paid more than their supervisor" and the model in Figure Three conforms to the policy. The policy is enforced by providing an OCL constraint and embedding that constraint in a validation framework. A constraint violation occurs when a domain modeler mistakenly assigns Michael to supervise Sam. Since Sam receives a salary of $20,000 and Michael, is paid $19,000 and that violates the salary policy. The system identifies Sam as the offending instance by selecting its graphical representation in the design pane and giving the user the ability to override the constraint before saving the model.

Visual studio's workbench provides a meta-model level validation framework that requires developers to design procedural C# methods that enforce OCL domain constraints. The C# procedures are embedded within a domain class and invoked when the domain model attempts to save a specific domain model.

The prototype supports translation of OCL to F# representations that are designed to work within a F# monadic workflow. Table one presents examples of semantic constraints, associated OCL and F# representations. The OCL constraints were represented in F# in a standard format that allows each constraint to either succeed or fail [8].

TABLE ONE.

Example of Translation from Text/OCL constraints to F# ready for monadic workflow processing

Text/ OCL Representation	F# Representation
Text: No employee shall be paid more than their supervisor OCL: Context Employee inv Salary Self.supervisees -> select (x| x.salary <= self. Salary)	**salaryLevelConstraint supervisedsalaries mySalary** = *if (**supervised**Salaries* |> *Seq.filter* (fun salary -> **salary > mySalary**) |> *Seq.isEmpty)* *then* *Success(("valid", "No employee is paid more than their supervisor"))* *else Failure({Message = " This salary must be less than supervisors"})*
Text: No employee shall be paid more than $30,000 OCL: context AllEmployees inv self.employees = forAll(s | s.salary <= 30000	**salaryManagerConstraint allEmployee**s = *if (**allEmployees*** |> *Seq.filter* (fun salary -> not (**salary <= 30000**) |> *Seq.isEmpty) then* *Success(("valid", " No employee is paid more than $30,000"))* *else* *Failure({Message = "This employee is paid more than $30,000"})*

In this manner, formatting constraints support the ability to layer workflows, where individual layers allow commutation of constraint groupings. This approach emphasizes scalability and maintainability by allowing constraints to be developed and tested by partitioning the space into related set of business domain constraints. The layers can then be combined contemporaneously to support and incrementally validate domain models. No changes to the monadic workflow framework are required to add or rearrange constraints.

Modularity and scalability are improved. For example, the salary constraint can be grouped into a workflow layer for processing:

```
salaryProcessor {
                  let! salary Result = salaryManagerConstraint allEmployees
                  let! maxSalaryResult = salaryLevelConstraint supervisees mySalary
                  let ! ...
                  let ! .. .
                  return AllResults,"valid"
                }
```

Within a workflow, constraints can be placed in any order and new constraints added or deleted without revision of the workflow's monadic framework. Within a layer, if any one constraint fails the entire workflow layer fails, otherwise the workflow layer is validated.

DISCUSSION AND CONCLUSION

Modeling standards specify OCL as a formal way to annotate policy constraints on UML diagrams. OCL language translators are required to transform OCL constraint into code structures that are incorporated into working systems. Similarly, OCL annotations within DSL meta-models require translation into code to apply the resulting translations in validating a domain model. Current tools rely on embedding customized validation methods and offer limited support in partitioning, composing and maintaining constraint groupings.

Developing DSL's for a business domain requires organization and maintenance of a complex set of business policy constraints. This study investigated a new design technique that transforms OCL annotations into declarative and functional forms and then applies those forms within a monadic framework. We translated OCL into F# representations and demonstrated a way to commute and compose groups of constraints into F# workflows. Composite workflows drive validation as modelers iterate with DSL tools to refine domain models.

Each constraint is defined separately as an F# function and placed into a scalable constraint library. As demonstrated, constraints can be aggregated into combinations or applied separately in any order. The modeler only needs to specify the constraints that apply and the monadic workflow structure processes the changes to validate the success or failure of workflows.

We demonstrated monadic structures that process composite workflows to validate a DSL domain model. Monadic structures are readily implemented in a higher-order, functional programming languages such as F# and offer a promising approach to create combinations of complex domain constraint networks in a manner that promotes modularity and scalability of DSL models.

REFERENCES

[1] Cabot, J. and E. Teniente, Transformation techniques for OCL constraints. Science of Computer Programming, 2007. 68(3), 152-168.

[2] Claessen, K., A poor man's concurrency monad. Journal of Functional Programming, 1999, 9(3), 313-323.

[3] Fowler, M., Domain Specific Languages. The Addison-Wesley Sgnature Series,2011, Boston, MA: Pearson Education, Inc.

[4] Hevner, A.R., et al., Design Science in Information Systems Research. MIS Quarterly, 2004, 28(1), 75-105.

[5] Kelly, S. and J.-P. Tolvanen, Domain-specific modeling: enabling full code generation,2008, New York, NY: Wiley-IEEE Computer Society Press.

[6] Marcondes, F.S., et al., Systematic and Formal Approach to got a Domain Specific Language, in *2009 Sixth International Conference on Information Technology: New Generations*,2009, IEEE: Las Vegas , NV. ,1447-1450.

[7] Requirements.net Consortium. The State of Business Analysis in Agile IT Projects. 2010; Available from: http://www.requirements.net/content/?id=63&register=done.

[8] Syme, D., A. Granicz, and A. Cisternino, Expert F# 2.0 (Expert's Voice in F#)2010, New York,N.Y., Springer-Verlag.

[9] Wadler, P., Comprehending monads, in *Mathematical Structures in Computer Science*1992, Cambridge University Press.,461-493.

[10] Wadler, P., How to declare an imperative. ACM Comput. Surv., 1997. 29(3), 240-263.