Specializing and Optimizing Declarative Domain Models

Srinivas Nedunuri and William Cook
Dept of Computer Sciences, University of Texas at Austin

Abstract In this position paper we propose specifying platform independent models using a functional language with a view to specializing and optimizing them using equational or algebraic reasoning. We illustrate our idea with the example of a very simple editor.

1. Introduction

Lately, as part of a general trend towards viewing "models as code", there has been growing interest in converting models from one form to another. For example, MDA from the OMG [Fr03] views the implementation process as one of converting a PIM (Platform Independent Model) to a PSM (Platform Specific Model). In transformational based development [CE00], the domain models are transformed via a series of intermediate domain models to a final implementation which is just a model in the target or implementation domain.

We suggest that the transformation of a PIM to a PSM can be viewed as a series of steps, as illustrated in Figure 1. At the top, generic models are reusable components that provide well-defined but generic functionality. A typical application will be a composition of multiple Generic PIMs [MB02] which define a subset of the domain model [Fr03]. In this paper we investigate the use of pure functional languages to formalize generic PIMs.

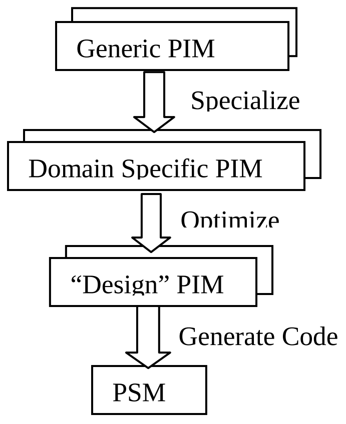
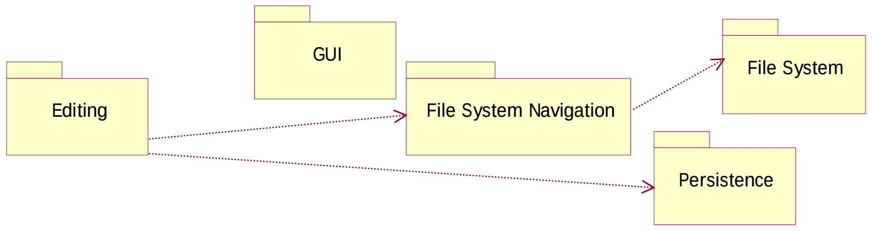


Figure 1: Steps in transforming a PIM to a PSM

The first transformation step specializes the generic PIMs with domain knowledge. This is analogous to the specialization of the Business Domain Layer to the Business Section Layer described in [BGK+97]. In the second step, we introduced the notion of platform-independent optimization as a transformation of the specialized PIMs into a model that includes design considerations such as specific data structures, algorithms, design patterns, architectural patterns, etc. The third step is the traditional transformation to platform specific code. The last two steps have their analogs in compilers, which first perform platform independent optimizations prior to invoking backend specific optimizers. This paper discusses the first two steps.

2. Generic PIMs

Consider the modeling of a simple generic Editor which contains one content model, a clipboard and a selection. An actual Editor application can be viewed as the composition of several PIMs as shown in the following domain chart:



In this paper we focus solely on the Editing domain. That is, we do not consider how model elements are presented on the screen (the View in the GUI domain) or the nature of the connection between the Editing and GUI domains (the Controller or bridge). A possible type model for the generic Editing PIM is shown in Figure 2.

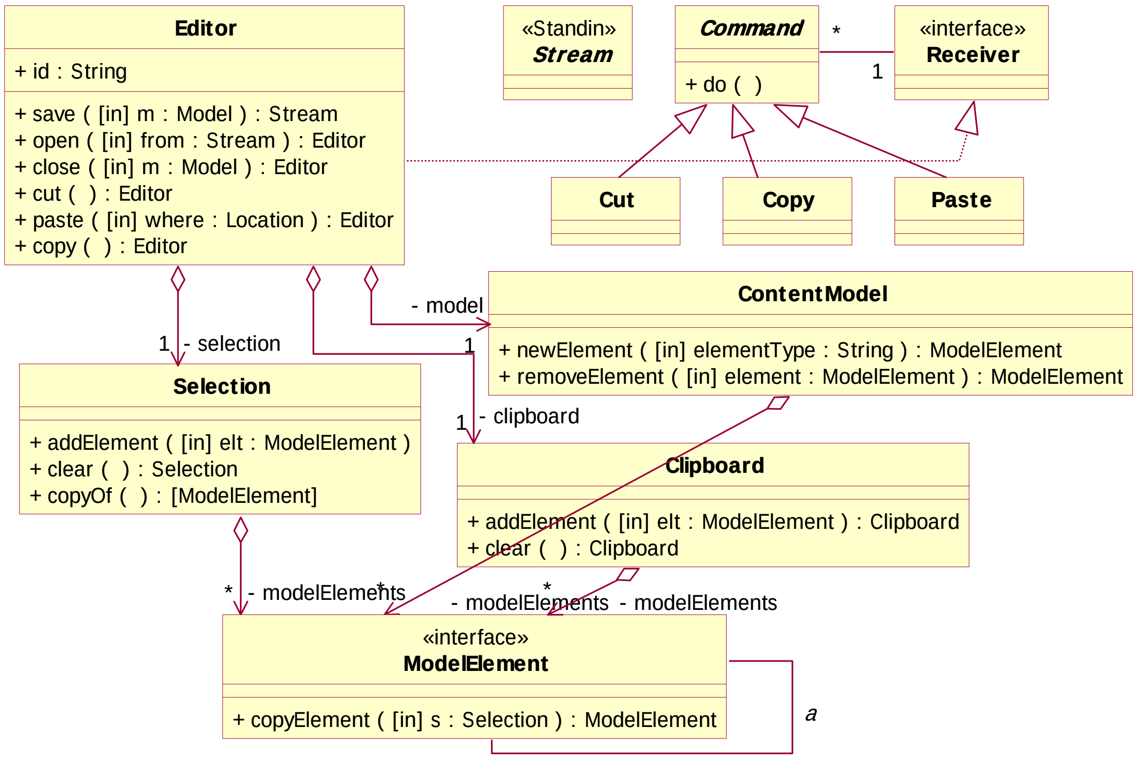

Editor
+ id : String
+ save ([in] m : Model) : Stream
+ open ([in] from : Stream) : Editor
+ close ([in] m : Model) : Editor
+ cut ( ) : Editor
+ paste ([in] where : Location) : Editor
+ copy ( ) : Editor
«Standin»
Stream
Command
+ do ( )
*
1
«interface»
Receiver
Cut
Copy
Paste
ContentModel
+ newElement ([in] elementType : String) : ModelElement
+ removeElement ([in] element : ModelElement) : ModelElement
- model
1
- selection
1
Selection
+ addElement ([in] elt : ModelElement)
+ clear ( ) : Selection
+ copyOf ( ) : [ModelElement]
1
- clipboard
Clipboard
+ addElement ([in] elt : ModelElement) : Clipboard
+ clear ( ) : Clipboard
*
- modelElements
*
- modelElements
*
- modelElements
«interface»
ModelElement
+ copyElement ([in] s : Selection) : ModelElement
a

Figure 2: UML Type diagram of Generic Editor Model

The generic domain model, analogous to a framework, defines the generic semantics of the Editing domain, which can be customized by the subdomains. The editor represents the information being edited as a `ContentModel` which contains a collection of `ModelElements`. `ModelElement` is an interface which must be supported by the specific types of model element being edited. The generic editor model supports abstractions for at least the cut, copy and paste commands. Our use of the Command pattern allows more commands to be added by the domain specific editors. The behavior of these operations is fully-specified by the generic model, but may also be specialized for particular applications. Although UML class diagrams are typically interpreted as representing object classes with mutable state, we apply a different interpretation to the classes as declarative data abstractions. Because our models are declarative (no side effects), the methods always return a new object, as indicated by the type declarations on the model above.

3. Why declarative models?

It has been suggested that avoiding side effects makes it easier to both develop correct programs [VAB00] and subsequently comprehend them [DLO+03]. Unfortunately, it is not easy to completely avoid side effects in most programming languages such as Java, C++, etc and still have an idiomatic efficient program. A related approach to developing correct programs known as Design by Contract [Me97] is to specify the program declaratively using a contract, and then write the body of the specification imperatively. However, it is rare to find a model that is fully specified using contracts. Part of the problem, we believe, is that after writing a declarative contract, one still has to write the imperative model, which increases, not decreases, the work effort. Functional languages take another approach to this problem: A functional program can be considered as a specification that is both side-effect free and executable. This gain can sometimes come at the cost of efficiency. Note, however, that in MDA, the idea is to develop the PIM with correctness, rather than efficiency, being the main criterion. Efficiency is addressed when transforming the PIM to a PSM. For this reason, we believe that functional language may be highly suitable for specifying the PIM. Functional languages also have a long history of research into program transformation [Da82], [BW89]. We hope to leverage off some of this work for transformation of the PIM to a PSM.

As a consequence of choosing a functional specification language, we can employ algebraic or equational transforms. That is, the expression (model) resulting from applying a transform is algebraically equal to the expression (model) to which it was applied. This very useful property enables us to view proofs as derivations and vice versa. That is, the resulting synthesized efficient model is correct by construction. Furthermore, because the derivations are also expressed in the same language, every intermediate stage in the derivation is also a valid program.

By choosing a specification language with powerful language capabilities such as higher order functions, lazy evaluation, type inference, pattern matching, etc. we hope to make the specification process itself more productive. Our first candidate specification language has been Haskell.

4. Declarative models in Haskell

Haskell is a modern functional programming language that is freely available [Th96]. In Haskell, types such as `Editor`, `Model` and others become abstract data types (ADTs) that are captured as modules in Haskell[1]. For example, the Editor module might be defined:

```
module Editor (Editor, save, open, close, copy, cut, paste) where
import Selection
import ContentModel
import Clipboard

data ModelElement modelElementT => Editor modelElementT =
       Editor {
            name :: String,
            model :: ContentModel modelElementT,
            selection :: Selection modelElementT,
            clipboard :: Clipboard modelElementT
            }

<… function definitions here …>
```

The ADT is defined inside a module, denoted by the keyword `module`. It exports the Editor type and the functions `save, open, close, copy, cut, paste`. The `imports` import all the dependent ADT modules, analogous to class imports in Java. The `data` part defines the fields or attributes of an Editor type. There are 4 fields, `name, model, selection, and clipboard`. The `modelElementT` is a type parameter, analogous to the type parameter that is used when defining generics in Java. The type parameter is qualified by requiring any specific type that instantiates the type parameter to be in the ModelElement typeclass (The OO analogue of this is requiring that the type supports the ModelElement interface). Thus, interfaces in the model become type classes in Haskell. Types in the ModelElement typeclass must have the copyElement function defined. This is stated as follows:

```
class ModelElement modelElementT where
   copyElement :: modelElementT -> Selection -> modelElementT
```

All of this mirrors fairly closely the type model for the Editing domain shown earlier.

As mentioned earlier, we ought to be able to use this framework to build a simple text editor and indeed we can do this by specializing the generic model

4.1. Domain Specialization of the Generic Model (Domain Specific Editors)

By specializing the generic Editing domain model shown earlier to particular domains (Step 1 in Fig. 1) we get domain specific editors. For example, in a logic circuit editor the ModelElements become `AND`, `OR`, `NAND` Gates, `Buses`, and `Timers`, there might be attributes such as `output`, `input1`, `input2`, etc. For a state model editor the ModelElements might be `State`, `Transition`, `Action` (perhaps subtyped as `EntryAction` and `ExitAction`), with attributes like `name`, `actions`, `guards`, `incoming`, and `outgoing`, `substates`, etc. A third example might be a text editor, with its metamodel shown in Figure 3.

There are two specializations on the model. The first is `Letter` implementing the `ModelElement` interface. The second is the association `a'` between `Letter` instances which is a specialization of the generic parent association `a` between model elements. (In general, there can be any number of specializations of the generic association).

[1] There are object-oriented declarative languages such as O Haskell [OH] which provide direct support for OO concepts such as methods and inheritance, while still retaining all the feature of functional programming. To simplify the presentation, we will just use ordinary Haskell in this paper. The core of our idea is not affected.

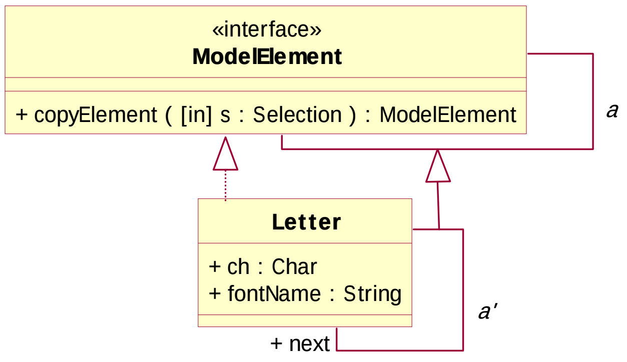


Figure 3: Specialization of generic model element for text

Of course, the above representation is quite unwieldy when it comes to text. We are not making use of the linearity property of text. That is, a `Letter` is associated with at most one other `Letter` (to its right). Put another way, if we were writing a text editor from scratch, it is unlikely that we would chose to represent the text as a graph of letter nodes. Instead a more direct linear form, such as an array or list is likely to be used. However, the rest of the model, which presumably provides functionality and services that we desire, not to mention all the other domains upon which this domain depends, expect to be able to interact with the model via the given graph based interface. This is achieved by defining an implementation of the Letter ADT that is based on lists, instead of graph nodes:

```
module TextAsList (Letter, singleLetter, newLetter, char, fontName, next,
set_next) where
import ObjectGraph
type Letter = [LetterData] -- the actual letter is the first elt in the list
-- it is done this way to create the impression of a graph of nodes.
data LetterData = LetterData {
                 m_char::Char,
                 m_fontName::String,
                } deriving Show
singleLetter :: Char -> String -> Letter
-- construct the actual letter and place it in a singleton list
singleLetter char fontname = [LetterData char fontname]

newLetter :: Char -> String -> Letter -> Letter
-- construct the actual letter and prefix to the front of the existing text
newLetter ch fontName next = (LetterData ch fontName) : next

char = m_char . head
fontName = m_fontName . head
next = tail
set_next letterData:_ nextLetter = letterData:nextLetter
```

In this implementation of the ADT, a `Letter` is defined as a list (sequence) of `LetterData` objects. In order that we can access the “next” letter in the sequence, the actual letter is just the first element in the list (`head`). The remainder of the list is the `tail` of the list. In order to get for example the `m_char` attribute, the corresponding accessor function, `char`, first accesses the `head` of the list, and then applies the built-in `m_char` accessor. This is written in Haskell using the function composition operator ‘.’. The `fontName` accessor is defined similarly. (Note that function application in Haskell is denoted by adjacency. That is, `f x` is `f` applied to `x`.) Because there is no destructive update, the “setters” (for example `set_next`) will just return a new instance. Two constructor methods are provided, `newLetter` and `singleLetter`. Both just turn around and call the automatically generated constructor “`Letter`”.

As an example of using the ADT interface, the model representing the string “cat” with ‘t’ in Times Roman and the rest in Arial font would be constructed with the following function calls:

```
newLetter 'c' "Arial" (newLetter 'a' "Arial" (singleLetter 't' "TimesRoman"))
```

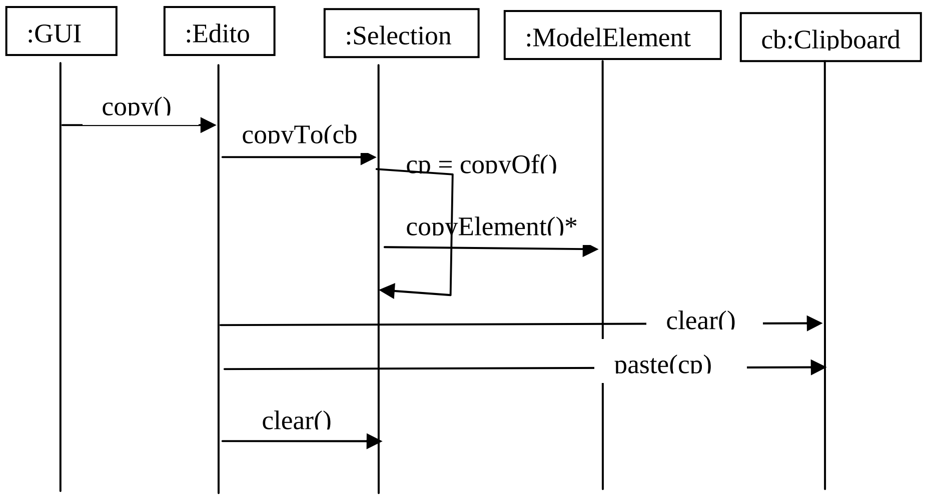
4.1.1. Proof of Correctness

In order to prove the correctness of the above implementation of a Letter ADT as a list we first define an implementation of the ADT which is a direct representation of the object graph. Then we use the fact that in a functional language, an expression such as the one shown above to construct the string "cat" is not just a just the sequence of calls to construct the string, but is also a term that represents the string itself. This allows us to induct over such expressions and show their equivalence in the two implementations. Space does not permit us to go into the details here. However, the proofs of this and the other transformations in this paper, as well as the complete executable code for the example are available from the authors. Having satisfied ourselves of the relationship between the external interface and internal representation, we can (with care) use the more direct list based representation when convenient.

Next we look at transformations that figure in the step of converting the PIM to a design PIM, which was identified as the 2nd step in Figure 1.

4.2. Optimizing the Specialized PIM to a Design PIM

As an example, consider the `copy` method in Editor. Below is a sequence diagram for the method:



A typical scenario for its usage is as follows: A user selects multiple diagram elements on the screen (perhaps by stretching a bounding box), the GUI Controller (part of the MVC pattern) repeatedly adds elements to the selection – using `selection.add(element)`. When done, the user might select the Copy menu command or use Ctl-C, or whatever other option the GUI provides. At that point, the controller calls `copy` on the editor, which then asks the selection for a copy of its elements (returned as a list, `cp`). Then the editor asks the clipboard to `clear` itself in preparation for pasting elements to it, and `paste` is called next. Finally, the selection is also asked to `clear` itself.

First we show the definition that is provided in Haskell as part of the generic PIM:

4.3. The Straightforward Definition of Selection

The goal of the copy method on Selection is to make a properly terminated copy of the graph of model elements contained in the selection. By "properly terminated" we mean that the associations to the other graph elements are only included if those other elements are themselves part of the selection. Below is (part of) the definition of the Selection ADT:

```
module Selection where
import Text -- or State or ...
type Selection modelElementT = [modelElementT]
newSelection = []
addElement element selection = element:selection
toList = id
```

This is just defining the `addElement` "method" as the builtin Haskell operation to prefix an element to a list (:); The `toList` method is just the identity function.

Now consider the definition of the `copyTo` function. Given a definition of an Element (like for example the ones for State, or Letter above), we can generate the `copyTo` function in the following way: Each element in the source is copied over including its primitive attributes. However, other elements to which the given element is associated are copied over only if they also lie in the source. This ensures that only the given subgraph is copied over[2].

A fully generic definition of copy requires the use of reflection or generic programming [HJ03, RJ03]. A function uses reflection to access meta-information: in this case `copyElement` needs access to the type information about the content model and its associations. Generic programming provides exactly this kind of polymorphism through the definition of type-indexed values: a function that includes generic cases for different kinds of types can be applied to any specific type. We have not yet applied generic programming in our implementation, but have instead created the required specialized `copyElement` function manually. The pattern for creation of the copy functions is given below:

```
copyTo src dest = dest ++ copyOf src

copyOf selection =
   [copyElement elt selection | elt <- theStronglyRootedElementsOf selection]

copyElement elt selection =
   let assoc1Val = confirm (assoc1 elt) selection
      assoc2Val = confirm (assoc2 elt) selection
      ...
      assocnVal = confirm (assocn elt) selection
   in
      elt{assoc1=assoc1Val, assoc2=assoc2Val,...,assocn=assocnVal}

confirm (Just val) set =
   if (val `elem` set) then Just (copyElement val set) else Nothing
confirm Nothing set = Nothing
```

`copyTo` appends a copy of the `src` onto the `dest` (using the list concatenation operator ++).

`copyOf` looks at each strongly rooted element (that is, elements in the object graph that do not have any association links between them. The roots however serve as the point to navigate to the rest of the connected component of the object graph) and calls copyElement on that object.

`copyElement` looks at the object at the other end of each association link (`assoc1` .. `assocn`) from the current object and copies it only if it is also contained in the selection. (`elem` is the membership function for a list.). `Just` and `Nothing` are data constructors belonging to the `Maybe` monad. They permit an association to either be empty or hold a value, somewhat analogous to the use of `null` vs. an object reference in Java.

Based on this template, we see that in the case where the `ModelElement` is a `Letter`, the specialization of `copyElement` for a Letter would be:

```
copyElement letter selection =
   let nextVal = if ((next letter) `elem` selection)
                 then copyElement (next letter) selection
                 else Empty
   in letter{next=nextVal}
```

4.4. The Optimized Definition of Selection

While this definition of Selection (and the contained method copy) by straightforward specialization of the PIM does the job, it is not the most efficient. If we know that a Selection of Letters in a text document must be contiguous (as it is in most text editors like Notepad or Word), there are some domain specific optimizations we can make. Firstly, because the selection is contiguous, there can only be one strongly rooted component. So when an element is added to the selection, we know that it is linked to the first (or last) element already in the selection. This permits us to eliminate a list of multiple components, and represent the selection by just two pieces of information: namely, the first letter of the selection and its length. The relevant code is shown below:

[2] To simplify the presentation, we are assuming an acyclic object graph. General graphs can be handled using Haskell's graph library module

```
data Selection = Selection{firstLetter::Letter, len::Int} deriving (Show, Eq)
_JUNK = (singleLetter 'x' "x")
newSelection = Selection _JUNK 0
addElement newSelection letter =
   selection{firstLetter = letter, len=length letter}
   where
      possUpdated1stLetter =
         if (len selection)== 0 -- if it’s a new selection
         then letter
              -- if the selection is being extended to the left
         else if (length letter) == (len selection) + 1
            then letter
            -- else its being extended to the right, don't change firstLetter
            else (firstLetter selection)
      newLength = (len selection) + 1 -- always update the length though!
```

This also means that `copyOf` has to deal with only one element:

```
copyTo selection dest = copyOf selection ++ dest
copyOf selection = [copyElement (firstLetter selection) selection]
```

Secondly, instead of a test and copy of each element in the chain, copyElement can just perform a block copy of the required letters:

```
copyElement letter selection = take (length selection) letter
```

where `take n` returns the first `n` elements of a list. This kind of optimization that crosses encapsulation boundaries is difficult to do in an imperative setting because of possibility of global state update. Note also that this optimized definition of the Selection ADT is algebraically derivable from the original (given the conditions stated above, namely that the ADT interface is restricted to singleton selections)

Finally as another example of domain specific optimization, if the text editor is a very simple one in which individual characters cannot differ in font (e.g. as in an editor like Notepad), we can eliminate the font attribute completely and replace `LetterData` in

```
type Letter = [LetterData]
```

by the character type (`Char`). In Haskell, `String = [Char]`, so we end up with

```
type Letter = String
```

Since we no longer have position information, we would of course revert back to the original definition of `Selection` as a list. Again it is possible to algebraically derive this transformation.

5. Future Work

The example presented in this paper was intended to provide a flavor of the approach we are investigating. Though the example is rooted in something “real world”, we have not addressed issues of scale and usage. For example, transformations of the type we have shown will need to be carried out many times in the process of mapping to an implementation. In order to capture design patterns as transformations, as suggested by Tokuda and Batory [TB95], we would need to be able to “chunk up” several derivation steps into one. This is where we think good proof assistants will come in useful. We would also like to be able to synthesize solutions rather than attempting to derive them after the fact.

Good tool support to carry out transformations and manage libraries of transformations is going to be essential. We plan to look at tools such as MAG [MS] and XT [JVV01] from the program transformation world.

We would also like to investigate “fusion” languages such as Scala [OAC04] and O Haskell [No99] that provide support for OO concepts such as classes and inheritance within a functional programming paradigm. This would make for a smoother conversion from OO models (defined for example in a declarative UML subset) to the specification language. We hope to be able to use one of the model transformation tools such as GMT[BEW03] for this purpose.

6. Related Work

Our work shares the same goals as much of the work in the fields of domain modeling and of MDA. The MIC group at Vanderbilt has been very active in the area of model transformation and aspect weaving. For example, [SAL+03] uses graph grammars to transform domain models. However, their emphasis is primarily on structural transformation. We believe that transforming behavior is at least as important as transforming structure. Also, they do not discuss how to prove or derive their transformations. Gray et.al. [GZL+04] apply transformation rules to legacy code using the DMS toolkit from Semantic Designs [SD]. However, their goal is reverse engineering rather than forward synthesis. [CH03] surveys and classifies the main transformational approaches, and also explains why intermediate models are useful.

The primary difference is in our use of declarative models, and of algebraic transforms on those models. Also, to the best of our knowledge, there has not been extensive work in the MDA arena on specializing generic models.

Functional programming has a long history of work in program transformation [Da82]. Almost every book on functional programming has a chapter or two on program synthesis [BW89], [Th96]. But, despite the fact that transformations have been studied quite extensively in functional programming, much of the attention has been focused on reducing algorithmic complexity and not on structural or model transformation. We want to leverage such work towards solving a slightly different problem.

There is also an extensive body of work in algebraic specification [Ba89], particularly dealing with proofs of ADT implementation. However, algebraic specifications are not in general executable. We believe that executability is very important for the purposes of validation.

7. References

[BGK+97] D. Baumer, G. Gryczan, R. Knoll, C Lilienthal, D Riehle, H Zullighoven, “Framework Development for Large Systems”, Communications of the ACM, V 40, #10, Oct. 1997.

[BW89] R. Bird, P. Wadler, Introduction to Functional Programming, Prentice Hall, 1989

[CE00] K. Czarnecki and E. Eisenecker, Generative Programming: Tools, Techniques, and Methods, Addison Wesley, 2001.

[CH03] K. Czarnecki, S. Helsen, “Classification of Model Translation Approaches”, Workshop on Generative Techniques in the context of Model Driven Architecture, OOPSLA, 2003

[Ba89] J. Baillie, “An Introduction to the Algebraic Specification of Abstract Data Types”, Dept. of Comp. Sci., Univ. of Hertfordshire, 1989.

[Da82] J. Darlington, “Program Transformation”, in Functional Programming and its Applications, Cambridge Univesity Press, 1982.

[DLO+03] J. J. Dolado, M. Harman, M. C. Otero, L. Hu, “An Empirical Investigation of the influence of a type of side effects on Program Comprehension”, IEEE Trans. On Software Engineering, V 29, #7, July 2003

[Fr03] D. Frankel, Model Driven Architecture, Addison-Wesley, 2003.

[GHJV94] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable OO Software, Addison Wesley, 1994

[GZL+04] J. Gray, et.al., “Model-driven Program Transformation of a Large Avionics Framework”.to appear in 3rd Intl. Conf. on Generative Programming and Component Engineering (GPCE 04), 2004

[HJ03] R. Hinze and J. Jeuring. "Generic Haskell: Practice and Theory". Lecture notes from Summer School on Generic Programming, LNCS 2793, Springer-Verlag, 2003.

[JVV01] M. de Jonge, E. Visser, J. Visser, XT: A bundle of program transformation tools, Electronic Notes in Theoretical Computer Sci., 44, #2, 2001

[Me97] B. Meyer, Object Oriented Software Construction, 2nd Ed, Prentice Hall, 1997

[MB02] Steve Mellor and Marc Balzer, Executable UML, Addison Wesley, 2002.

[MS99] O. de Moor and G. Sittampalam, “Generic Program Transformation”, Proc. 3rd Intl. Summer School on Advanced Functional Programming, LNCS v.1608, Springer Verlag, 1999

[No99] Nordlander,“Reactive Objects and Functional Programming”, PhD Thesis, Dept. of Comp. Sci., Chalmers Univ. of Tech. Sweden, 1999

[RJ03] R. Lammel and S. Peyton Jones. *Scrap your boilerplate: a practical approach to generic programming*. In ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'03), 2003, pages 26-37.

[SAL+03] J. Sprinkle, A. Agrawal, T. Levendovszky, F Shi, G Karsai, Domain Model Translation Using Graph Transformations, 10th IEEE International Conf. and Workshop on the Engineering of Computer Based Systems, 2003.

[OAC+04] M. Odersky et.al., “An Introduction to Scala”, Programming Methods Laboratory, EPFL, Switzerland, June 2004

[SD] Design Maintenance System, Semantic Designs, www.semdesigns.com.

[TB95] L. Tokuda, D. Batory, Automating Software Evolution via Design Pattern Transformations, Univ. of Texas at Austin, 1995.

[Th96] S. Thompson, Haskell:The Craft of Functional Programming, Addison-Wesley, 1996.

[BEW03] J. Bettin, G. van Emde Boas, E. Willink, “Generative Model Transformer: An Open Source MDA Tool Initiative”, OOPSLA, 2003

[VAB+00] Vermuelen, Ambler, Bumgardner, Metz, Misfeldt, Shur, Thompson, Elements of Java Style, Cambridge Univesity Press, 2000.