Model Transformation Languages Relying on Models as ADTs

Jerónimo Irazábal and Claudia Pons

LIFIA, Facultad de Informática, Universidad Nacional de La Plata
Buenos Aires, Argentina
{jirazabal,cpons}@lifia.info.unlp.edu.ar

Abstract. In this paper we describe a simple formal approach that can be used to support the definition and implementation of model to model transformations. The approach is based on the idea that models as well as metamodels should be regarded as abstract data types (ADTs), that is to say, as abstract structures equipped with a set of operations. On top of these ADTs we define a minimal, imperative model transformation language with strong formal semantics. This proposal can be used in two different ways, on one hand it enables simple transformations to be implemented simply by writing them in any ordinary programming language enriched with the ADTs. And on the other hand, it provides a practical way to formally define the semantics of more complex model transformation languages.

Keywords: Model driven software engineering, Model transformation language, denotational semantics, Abstract data types, ATL.

1 Introduction

Model-to-model transformations are at the core of the Model Driven Engineering (MDE) approach [1] and it is expected that writing such transformations will become an ordinary task in software development. Specification and implementation of model-to-model transformation involves significant knowledge of both the source and target domains. Even when the transformation designer understands both domains, defining the mapping between corresponding model elements is a very complex task.

One direction for reducing such complexity is to develop domain-specific languages designed to solve frequent model transformation tasks. A domain-specific language focuses on a particular problem domain and contains a relatively small number of constructs that are immediately identifiable to domain experts. Domain-specific languages allow developers to create abstract, concise solutions in a simpler way.

Indeed, this is the approach that has been taken by the MDE community. As a result, a variety of model transformation domain-specific languages have been recently developed, e.g. QVT [2], ATL [3], Tefkat [4] and VIATRA [5]. These languages are very rich and are used in various domains; each of them possesses its own syntax, programming paradigm and other specific language peculiarities.

However, there are a number of facts that frequently hinder the use in industry of these specific languages, on one hand their application require a large amount of

M. van den Brand, D. Gašević, J. Gray (Eds.): SLE 2009, LNCS 5969, pp. 133–143, 2010.

learning time that cannot be afforded in most projects; on the other hand considerable investment in new tools and development environments is necessary. And finally, the semantics of these specific languages is not formally defined and thus, the user is forced to learn such semantics by running transformations example suites within a given tool. Unfortunately, in many cases the interpretation of a single syntactic construct varies from tool to tool. Additionally other model engineering instruments, such as mechanism for transformation analysis and optimization, can be only built on the basis of a formal semantics for the transformation language; therefore, a formal semantics should be provided.

To overcome these problems, in this paper we describe a minimal, imperative approach with strong formal semantics that can be used to support the definition and implementation of practical transformations. This approach is based on the idea of using "models as abstract data types" as the basis to support the development of model transformations. Specifically, we formalize models and metamodels as abstract mathematical structures equipped with a set of operations. The use of this approach enables transformations to be implemented in a simpler way by applying any ordinary imperative programming language enriched with the ADTs, thus avoiding the need of having a full model transformation platform and/or learning a new programming paradigm. Additionally, the meaning of the transformation language expressions is formally defined, enabling the validation of transformation programs.

Complementary, this approach offers an intermediate abstraction level which provides a practical way to formally define the semantics of higher level model transformation languages

The paper is organized as follows. Section 2 provides the formal characterization of models and metamodels as abstract mathematical structures equipped with a set of operations. These mathematical objects are used in section 3 for defining the semantics of a basic transformation language. Section 4 illustrates the use of the approach to solve a simple transformation problem, while Section 5 shows the application of the approach to support complex transformation languages (in particular ATL). Section 6 compares this approach with related research and Section 7 ends with the conclusions.

2 Model Transformation Languages with ADTs

A model transformation is a program that takes as input a model element and provides as output another model element. Thinking about the development of this kind of programs there are a number of alternative ways to accomplish the task:

A very basic approach would be to write an ordinary program containing a mix of loops and *if* statements that explore the input model, and create elements for the output model where appropriate. Such an approach would be widely regarded as a bad solution and it would be very difficult to maintain.

An approach situated on the other extreme of the transformation language spectrum would be to count with a very high level declarative language specially designed to write model transformations (e.g. QVT Relations [2]). With this kind of language we would write the 'what' about the transformation without writing the 'how'. Thus, neither concrete mechanism to explore the input model nor to create the output model is exposed in the program. Such an approach is very elegant and concise, but the

meaning of the expressions composing these high level languages becomes less intuitive and consequently hard to understand. In addition, the implementation of heavy supporting framework is required (e.g. MediniQVT supporting tools [6]).

A better solution, from a programming perspective, would be to build an intermediate abstraction level. We can achieve this goal by making use of abstract data types to structure the source and target models. This solution provides a controlled way to traverse a source model, and a reasonable means to structure the code for generating an output model. With this solution we would raise the abstraction level of transformation programs written in an ordinary programming language, while still keeping the control on the model manipulation mechanisms. Additionally we do not need to use a new language for writing model transformations, since any ordinary programming language would be sufficient.

Towards the adoption of the later alternative, in this section we formally define the concepts of model and metamodel as *Abstract Data Types* (ADTs), that is to say as abstract structures equipped with a set of operations.

***Definition 1*: A metamodel** is a structure mm = (C, A, R, *s*, *a*, *r*) where C is the set of classes, A is the set of attributes and R is the set of references respectively; *s* is an anti-symmetric relation over C interpreted as the superclass relation, *a* maps each attribute to the class it belongs to; and *r* maps each reference to its source and target classes.

For example, a simplified version of the Relational Data Base metamodel is defined as MMRDB =(C, A, R, *s*, *a*, *r*), where:

C={Table, Column, ForeignKey}
A={$name_{Table}$, $name_{Column}$}
R={$columns_{Table2Column}$, $primaryKey_{Table2Column}$, $foreignKeys_{Table2ForeignKey}$, $table_{ForeignKey2Table}$}
s= {} a= {($name_{Table}$, Table), ($name_{Column}$, Column)}
r= {($columns_{Table2Column}$, (Table, Column)), ($primaryKey_{Table2Column}$, (Table, Column)), ($foreignKeys_{Table2ForeignKey}$, (Table, ForeignKey)), ($table_{ForeignKey2Table}$, (ForeignKey, Table))}

The usual way to depict a metamodel is by drawing a set of labeled boxes connected by labeled lines; however the concrete appearance of the metamodel is not relevant for our purposes. Figure 1 shows the simplified metamodel of the Relational Data Base language.

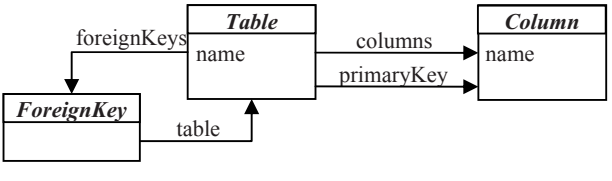

Fig. 1. The metamodel of the Relational Data Base language

For the sake of simplicity, we assume single-valued single-typed attributes and references without cardinality specification. The previous metamodel definition could be easily extended to support multi-valued multi-typed attributes and to allow the specification of reference's cardinality; however, in this paper those features would only complicate our definitions, hindering the understanding of the core concepts.

***Definition 2*: A model** is a structure m = (C, A, R, *s*, *a*, *r*, E, *c*, *va*, *vr*) where mm = (C, A, R, *s*, *a*, *r*) is a metamodel, E is the set of model elements, *c* maps each element to the class it belongs to, *va* applied to an attribute and to an element returns the value of such attribute (or bottom, if the attribute is undefined), *vr* applied to a reference and an element returns a set of elements that are connected on the opposite end of the reference. In such case, we say that model m is an instance of metamodel mm. When the metamodel is obvious from the context we can omit it in the model structure.

For example, let m = (C, A, R, *s*, *a*, *r*, E, *c*, *va*, *vr*) be an instance of the MMRDB metamodel, where:

mm = (C, A, R, s, a, r) is the metamodel defined above,
E={Book, Author, $name_{Book}$, $editorial_{Book}$, $authors_{Book2Author}$, $name_{Author}$}
c={(Book, Table), (Author, Table), ($name_{Book}$, Column), ($editorial_{Book}$, Column), ($authors_{Book2Author}$, ForeignKey), ($name_{Author}$, Column)}
va={(($name_{Table}$, Book), Book), (($name_{Table}$, Author), Author)}
vr= {(($columns_{Table2Column}$, Book), {$name_{Book}$, $editorial_{Book}$}) , (($columns_{Table2Column}$, Author), {$name_{Author}$}), (($primaryKey_{Table2Column}$, Book), {$name_{Book}$}), (($primaryKey_{Table2Column}$, Author), {$name_{Author}$}), (($foreignKeys_{Table2ForeignKey}$, Book), {$authors_{Book2Author}$}), (($table_{ForeignKey2Table}$, $authors_{Book2Author}$), {Book})})

Figure 2 illustrates the instance of the MMRDB metamodel in a generic graphical way. The concrete syntax of models is not relevant here.

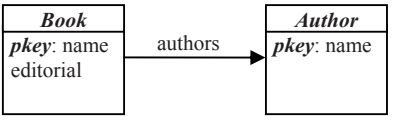

Fig. 2. An instance of the MMRDB metamodel

After defining the abstract structure of models and metamodels we are ready to define a set of operations on such structure. These operations complete the definition of the Abstract Data Type.

Let **M** be the set of models and let **MM** be the set of metamodels, as defined above. The following functions are defined:

(1) The function **metamodel()** returns the metamodel of the input model.

metamodel: M → MM

metamodel (**C**, **A**, **R**, ***s***, ***a***, ***r***, E, *c*, *va*, *vr*) = (**C**, **A**, **R**, ***s***, ***a***, ***r***)

(2) The function **classOf()** returns the metaclass of the input model element in the context of a given model.

classOf: E → M → C

classOf e (**C**, **A**, **R**, ***s***, ***a***, ***r***, E, *c*, *va*, *vr*) = c(e)

(3) The function **elementsOf()** returns all the instances of the input class in the context of a given model. Instances are obtained by applying the inverse of function c.
elementsOf: $C \rightarrow M \rightarrow P(E)$
elementsOf c (**C**, **A**, **R**, ***s***, ***a***, ***r***, E, *c*, *va*, *vr*) = $c^{-1}(c)$
(4) The function **new()** creates a new instance of the input class and inserts it into the input model.
new: $C \rightarrow M \rightarrow ExM$
new c (**C**, **A**, **R**, ***s***, ***a***, ***r***, E, *c*, *va*, *vr*) =
(e, (**C**, **A**, **R**, ***s***, ***a***, ***r***, $E \cup \{e\}$, *c*[e←c], *va*, *vr*)), with $e \notin E$
(5) The function **delete()** eliminates the input element from the input model.
delete: $E \rightarrow M \rightarrow M$
delete e (**C**, **A**, **R**, ***s***, ***a***, ***r***, E, *c*, *va*, *vr*) = (**C**, **A**, **R**, ***s***, ***a***, ***r***, E', *c'*, *va'*, *vr'*),
with E' = E-{e}, *c'* = *c*-{(e,c(e))},
va' = *va*-{(a,(e',n))| e=e' ^ (a,(e',n)) ∈ *va*}
vr' = *vr*-{(r,(e',es))| e=e' ^ (r,(e',es)) ∈ *vr*}
(6) The function **getAttribute()** returns the value of the input attribute in the input element belonging to the input model.
getAttribute: $A \rightarrow E \rightarrow M \rightarrow Z_{\perp}$
getAttribute a e (**C**, **A**, **R**, ***s***, ***a***, ***r***, E, *c*, *va*, *vr*) = *va*(a)(e)
(7) The function **setAttribute()** returns an output model resulting from modifying the value of the input attribute in the input element of the input model.
setAttribute: $A \rightarrow E \rightarrow Z_{\perp} \rightarrow M \rightarrow M$
setAttribute a e n (**C**, **A**, **R**, ***s***, ***a***, ***r***, E, *c*, *va*, *vr*) =
(**C**, **A**, **R**, ***s***, ***a***, ***r***, E, *c*, *va*[*va*(a)[e←n]], *vr*), if (a,*s*(e)) ∈ ***a***
(**C**, **A**, **R**, ***s***, ***a***, ***r***, E, *c*, *va*, *vr*), if (a,ς(e)) ∉ ***a***
(8) The function **getReferences()** returns the set of elements connected to the input element by the input reference in the input model.
getReferences: $R \rightarrow E \rightarrow M \rightarrow P(E)$
getReferences r e (**C**, **A**, **R**, ***s***, ***a***, ***r***, E, *c*, *va*, *vr*) = *vr*(r)(e)
(9) The function **addReference()** returns an output model resulting from adding a new reference (between the two input elements) to the input model.
addReference: $R \rightarrow E \rightarrow E \rightarrow M \rightarrow M$
addReference r e e' (**C**, **A**, **R**, ***s***, ***a***, ***r***, E, *c*, *va*, *vr*) =
(**C**, **A**, **R**, ***s***, ***a***, ***r***, E, *c*, *va*, *vr* ∪(r,(e,e'))), if (r,(*c*(e), *c*(e'))) ∈ ***r***
(**C**, **A**, **R**, ***s***, ***a***, ***r***, E, *c*, *va*, *vr*), if (r,(*c*(e), *c*(e'))) ∉ ***r***
(10) The function **removeReference()** returns an output model resulting from deleting the input reference between the two input elements from the input model.
removeReference: $R \rightarrow E \rightarrow E \rightarrow M \rightarrow M$
removeReference r e e' (**C**, **A**, **R**, ***s***, ***a***, ***r***, E, *c*, *va*, *vr*) =
(**C**, **A**, **R**, ***s***, ***a***, ***r***, E, *c*, *va*, *vr* -(r,(e,e')))

The remaining functions (e.g. similar functions, but on the metamodel level) are omitted in this paper for space limitations.

3 A Simple Yet Powerful Imperative Transformation Language

In this section we define SITL a simple imperative transformation language that supports model manipulation. This language is built on top of a very simple imperative language with assignment commands, sequential composition, conditionals, and finite iterative commands. As a direct consequence, this language has a very intuitive semantics determined by its imperative constructions and by the underlying model ADT.

This language is not intended to be used to write model transformation programs, rather it is proposed as a representation of the minimal set of syntactic constructs that any imperative programming language must provide in order to support model transformations. In practice we will provide several concrete implementations of SITL. Each concrete implementation consists of two elements: an implementation of the ADTs and a mapping from the syntactic constructs of SITL to syntactic constructs of the concrete language.

3.1 Syntax

The abstract syntax of STIL is described by the following abstract grammar:

<intexp> ::= **null** | 0 | 1 | 2 | …
| <intvar> | - <intexp> | <intexp> + <intexp> | <intexp> - <intexp>
| <intexp> * <intexp> | <intexp> ÷ <intexp>| <elemexp> . <attrexp>
| **size** <elemlistexp>
<boolexp> ::= **true** | **false**
| <intexp> = <intexp> | <intexp> < <intexp> | <intexp> > <intexp>
| ¬ <boolexp> | <boolexp> ∧ <boolexp> | <boolexp> ∨ <boolexp>
| <elemexp> = <elemexp> | **contains** <elemlistexp> <elemexp>
<modelexp> ::= m_1 | m_2 | ...
<classexp> ::= c_1 | c_2 | …| **classof** <elemexp>
<attrexp> ::= a_1 | a_2 | …
<refexp> ::= r_1 | r_2 | …
<elemexp> ::= <elemvar> | <elemlistexp> (<intexp>)
<elemlistexp> ::= **elementsOfClass** <classexp> **inModel** <modelexp>
| <elemlistvar> | <elemexp> . <refexp>
<comm> ::= <intvar> := <intexp> **in** <comm> | <elemvar> := <elemexp> **in** <comm>
| <elemlistvar> := <elemlistexp> **in** <comm> | <comm> ; <comm>
| **skip** | **if** <boolexp> **then** <comm> **else** <comm>
| **for** <intvar> **from** <intexp> **to** <intexp> **do** <comm>
| **add** <elemexp> **to** < elemlistvar> | **remove** <elemexp> **from** <elemlistvar>
| <elemexp> . <attrexp> := <intexp>
| **addRef** <elemexp> <refexp> <elemexp>
| **removeRef** <elemexp> <refexp> <elemexp>
| **forEachElem** <elemvar> **in** <elemlistexp> **where** <boolexp> **do**<comm>
| **newElem** <elemvar> **ofclass** <classexp> **inModel** <modelexp>
| **deleteElem** <elemexp>

<procD>::=**proc** <name> <procparams> **beginproc** <comm> **endproc**| <procD>;<procD>
<comm'> ::= <comm> | **call** <name> (actualparams) | <comm'> ; <comm'>
<program>::= <procD> <comm'>

Currently, we consider three types of variables: integer variables, element variables, and list of elements variables. It is worth to remark that STIL is limited to finite programs; we argue that model to model transformation should be finite, so this feature is not restrictive at all.

Denotational Semantics

The semantics of SITL is defined in the standard way [7]; we define semantic functions that map expressions of the language into the meaning that these expressions denote. The usual denotation of a program is a state transformer. In the case of SITL each state holds the current value for each variable and a list of the models that can be manipulated. More formally, a **program state** is a structure $\sigma = (\sigma_M, \sigma_{EM}, \sigma_E, \sigma_{Es}, \sigma_Z)$ where σ_M is a list of models, σ_{EM} maps each element to the model it belongs to, σ_E maps element variables to elements, σ_{Es} maps element list variable to a list of elements, and σ_Z maps integer variables to its integer value or to bottom.
Let Σ denote the set of program states; the semantic functions have the following signatures:

$[[-]]_{intexp} : <intexp> \rightarrow \Sigma \rightarrow Z_\perp$
$[[-]]_{boolexp} : <boolexp> \rightarrow \Sigma \rightarrow B$
$[[-]]_{modelexp} : <modelexp> \rightarrow \Sigma \rightarrow M_\perp$
$[[-]]_{classexp} : <classexp> \rightarrow \Sigma \rightarrow C$
$[[-]]_{elemexp} : <elemexp> \rightarrow \Sigma \rightarrow E$
$[[-]]_{elemlistexp} : <elemlistexp> \rightarrow \Sigma \rightarrow [E]$
$[[-]]_{attrexp} : <attrexp> \rightarrow \Sigma \rightarrow Z$
$[[-]]_{refexp} : <refexp> \rightarrow \Sigma \rightarrow Z$
$[[-]]_{comm} : <comm> \rightarrow \Sigma \rightarrow \Sigma$

Then, we define theses functions by semantic equations. The semantic equations of Integer expressions and Boolean expressions are omitted, as well as some equations related to well-understood constructs such as conditionals, sequences of commands and procedure calls. For the following equations let $\sigma = (\sigma_M, \sigma_{EM}, \sigma_E, \sigma_{Es}, \sigma_Z) \in \Sigma$:

– Equations for integer expressions

$[[\text{null}]]_{intexp}\ \sigma = \perp$

$[[\text{e . a}]]_{intexp}\ \sigma = \textit{getAttribute}\ ([[\text{a}]]_{attrexp}\ \sigma)\ ([[\text{e}]]_{elemexp}\ \sigma)\ (\sigma_M\ (\sigma_{EM}\ ([[\text{e}]]_{elemexp}\ \sigma)))$

– Equations for class expressions

$[[\textbf{classof}\ \text{e}]]_{classexp}\ \sigma = \textit{classOf}\ ([[\text{e}]]_{elemexp}\ \sigma)\ (\sigma_M\ (\sigma_{EM}\ ([[\text{e}]]_{elemexp}\ \sigma)))$

– Equations for element expressions

$[[e_x]]_{elemexp}\ \sigma = \sigma_E\ (e_x)$

– Equations for element list expressions

$[[\textbf{elementsOfClass}\ \text{c}\ \textbf{inModel}\ \text{m}]]_{elemlistexp}\ \sigma =$
$\textit{elementsOf}\ ([[\text{c}]]_{classexp}\ \sigma)\ (\sigma_M\ ([[\text{m}]]_{modelexp}\ \sigma))$

$[[es_x]]_{elemlistexp}\ \sigma = \sigma_{Es}\ (es_x)$

$[[e\ .\ r]]_{elemlistexp}\ \sigma = \mathit{getReferences}\ ([[r]]_{refexp}\ \sigma)\ ([[e]]_{elemexp}\ \sigma)\ (\sigma_M\ (\sigma_{EM}\ ([[e]]_{elemexp}\ \sigma)))$

– Equations for commands

$[[x := ie]]_{comm}\ \sigma = (\sigma_M, \sigma_{EM}, \sigma_E, \sigma_{Es}, \sigma_Z[x \leftarrow ([[ie]]_{intexp}\ \sigma)])$

$[[e_x := ee]]_{comm}\ \sigma = (\sigma_M, \sigma_{EM}, \sigma_E[e_x \leftarrow ([[ee]]_{elemexp}\ \sigma)], \sigma_{Es}, \sigma_Z)$

$[[e\ .\ a := ie]]_{comm}\ \sigma =$
$\mathit{setattribute}\ ([[a]]_{attrexp}\ \sigma)\ ([[e]]_{elemexp}\ \sigma)\ ([[ie]]_{intexp}\ \sigma)\ (\sigma_M\ (\sigma_{EM}\ ([[e]]_{elemexp}\ \sigma)))$
$[[\textbf{newElem}\ e_x\ \textbf{ofclass}\ c\ \textbf{inModel}\ m]]_{comm}\ \sigma = (\sigma_M', \sigma_{EM}', \sigma_E', \sigma_{Es}, \sigma_Z)$
with $im = [[m]]_{modelexp}\ \sigma$, $(e,m) = \mathit{new}\ ([[c]]_{classexp}\ \sigma)\ (\sigma_M\ (im))$,
$\sigma_M' = \sigma_M[im \leftarrow m]$, $\sigma_E' = \sigma_E[e_x \leftarrow e]$, $\sigma_{EM}' = \sigma_{ME}[e \leftarrow im]$
$[[\textbf{deleteElem}\ e]]_{comm}\ \sigma = (\sigma_M', \sigma_{EM}', \sigma_E, \sigma_{Es}, \sigma_Z)$
with $e' = [[e]]_{elemexp}\ \sigma$, $im = \sigma_{EM}\ e'$, $m = \mathit{delete}\ e'\ (\sigma_M\ im)$
$\sigma_M' = \sigma_M[im \leftarrow m]$, $\sigma_{EM}' = \sigma_{ME}[e \leftarrow im]$

$[[\textbf{for}\ x\ \textbf{from}\ ie_1\ \textbf{to}\ ie_2\ \textbf{do}\ c]]_{comm}\ \sigma = \textbf{\textit{iSec}}\ ([[ie_1]]_{intexp}\ \sigma)\ ([[ie_2]]_{intexp}\ \sigma)\ x\ c\ \sigma$
$\textbf{\textit{iSec}}\ n\ m\ x\ c\ \sigma = \sigma$, if $n > m$
$\textbf{\textit{iSec}}\ (n+1)\ m\ x\ c\ ([[c]]_{comm}\ ((\sigma_M, \sigma_{EM}, \sigma_E, \sigma_{Es}, \sigma_Z[x \leftarrow n])))$, if $n \le m$
$[[\textbf{forEachElem}\ e_x\ \textbf{in}\ es\ \textbf{where}\ b\ \textbf{do}\ c]]_{comm}\ \sigma = \textbf{\textit{eSec}}\ ([[es]]_{elemlistexp}\ \sigma)\ e_x\ b\ c\ \sigma$
$\textbf{\textit{eSec}}\ es\ e_x\ b\ c\ \sigma = \sigma$, if $es = \varnothing$
$\textbf{\textit{eSec}}\ es'\ e_x\ b\ c\ \sigma''$, if $es \neq \varnothing$
with $es = e:es'$, $\sigma' = (\sigma_M, \sigma_{EM}, \sigma_E[e_x \leftarrow e], \sigma_{Es}, \sigma_Z)$
$\sigma'' = [[c]]_{comm}\ \sigma'$, if $[[b]]_{boolexp}\ \sigma'$, $\sigma'' = \sigma'$, if not $[[b]]_{boolexp}\ \sigma'$

By applying these definitions we are able to prove whether two programs (i.e. transformations) are equivalent.

Definition 3: two programs t and t' are equivalent if and only if $([[t]]_{comm}\ \sigma)\sigma_M = ([[t']]_{comm}\ \sigma)\sigma_M$, for all $\sigma \in \Sigma$.

Note that this definition does not take the values of variables into consideration, so two programs using different sets of internal variables would even be equivalent. Equivalence is defined considering only the input and output models (observable equivalence).

4 A Simple Example

Let mm be the metamodel defined in section 2; let m1 be an instance of mm and m2 be the empty instance of mm. The following SITL program when applied to a state containing both the model m1 and the model m2 will populate m2 with the tables in m1, but none of the columns, primary keys or foreign keys will be copied to m2.

```
forEachElem t in (elementsOfClass Table inModel m1)
where true do
    newElem t' ofClass Table inModel m2; t'.name = t.name;
```

The resulting model is m2=(E, *c*, *va*, *vr*) where, E={Book, Author}, c={(Book, Table), (Author, Table)}, *va*={((name$_{Table}$, Book), *Book*), ((name$_{Table}$, Author), *Author*)}, *vr* = ∅).

A formal proof of the correctness of this transformation would be written in a straightforward way by using the SITL's semantics definition.

5 Encoding ATL in SITL

Due to the fact that SITL is situated at midway between ordinary programming languages and transformation specific languages, such intermediate abstraction level makes it suitable for being used to define the semantics of more complex transformation languages. With the aim of showing an example in this section we sketch how to encode ATL in SITL. Each ATL rule is encode into a SITL procedure.
Lets considerer the following simple rule template in ATL:

```
module m from m1: MM1 to m2: MM2
rule rule_name {
from
in_var1 : in_class1!MM1 (condition1),
…
in_varn : in_classn!MM1 (conditionn)
to
out_var1 : out_class1!MM2 (bindings1),
…
out_varm : out_classm!MM2 (bindingsm)
do {statements}}
```

The equivalent code fragment in SITL would be:

```
proc rule_name () beginproc
 forEachElem in_var1 in (elementsOfClass in_class1 inModel
m1)
 where condition1 do
 …
 forEachElem in_varn in(elementsOfClass in_classn inModel
m1)
 where conditionn do
     newElem out_var1 ofclass out_class1 inModel m2;
     …
     newElem out_varm ofclass out_classm inModel m2;
     bindings1;
     …
     bindingsm;
     statements;
endproc
```

A more complete encoding of ATL in SITL taking into account called rules and lazy unique rules can be read in [8].

6 Related Work

Sitra [9] is a minimal, Java based, library that can be used to support the implementation of simple transformations. With a similar objective, RubyTL [10] is an extensible transformation language embedded in the Ruby programming language. These proposals are related to ours in the sense that they aim at providing a minimal and familiar transformation framework to avoid the cost of learning new concepts and tools. The main difference between these works and the proposal in this paper is that we are not interested in a solution that remains confined to a particular programming language, but rather in a language-independent solution founded on a mathematical description.

Barzdins and colleagues in [11] define L0, a low level procedural strongly typed textual model transformation language. This language contains minimal but sufficient constructs for model and metamodel processing and control flow facilities resembling those found in assembler-like languages and it is intended to be used for implementation of higher-level model transformation languages by the bootstrapping method. Unlike our proposal, this language does not have a formal semantics neither is based on the idea of models as ADTs.

Rensink proposes in [12] a minimal formal framework for clarifying the concept of model, metamodel and model transformation. Unlike that work, our formal definitions are more understandable while still ensuring the characterization of all relevant features involved in the model transformation domain. Additionally, the proposal in [12] does not define a particular language for expressing transformations.

On the other hand, due to the fact that SITL is situated at midway between ordinary programming languages and transformation specific languages, such intermediate abstraction level makes it suitable for being used to define the semantics of complex transformation languages. In contrast to similar approaches - e.g. the translation of QVT to OCL+Alloy presented in [13] or the translation of QVT to Colored Petri Nets described in [14] - our solution offers a significant reduction of the gap between source and target transformation languages.

7 Conclusions

In this paper we have proposed the use of “models as abstract data types” as the basis to support the development of model transformations. Specifically, we have formalized models and metamodels as abstract mathematical structures equipped with a set of operations. This abstract characterization allowed us to define a simple transformation approach that can be used to support the definition and implementation of model-to-model transformations. The core of this approach is a very small and understandable set of programming constructs.

The use of this approach enables transformations to be implemented in a simpler way by applying any ordinary imperative programming language enriched with the ADTs, thus we avoid the overhead of having a full model transformation platform and/or learning a new programming paradigm.

Additionally, the meanings of expressions from the transformation language are formally defined, enabling the validation of transformation specifications. Such meaning is abstract and independent of any existing programming language.

Finally, we have shown that other well-known model transformation languages, such as ATL, can be encoded into this frame. Thus, this approach provides a practical way to formally define the semantics of complex model transformation languages.

References

[1] Stahl, T., Völter, M.: Model-Driven Software Development. John Wiley & Sons, Ltd., Chichester (2006)

[2] QVT Adopted Specification 2.0 (2005), http://www.omg.org/docs/ptc/05-11-01.pdf

[3] Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)

[4] Lawley, M., Steel, J.: Practical Declarative Model Transformation With Tefkat. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 139–150. Springer, Heidelberg (2006)

[5] Varro, D., Varro, G., Pataricza, A.: Designing the Automatic Transformation of Visual Languages. Science of Computer Programming 44(2), 205–227 (2002)

[6] Medini, QVT. ikv++ technologies ag, http://www.ikv.de (accessed in December 2008)

[7] Hennessy's Semantics of Programming Languages. Wiley, Chichester (1990)

[8] Irazabal, J.: Encoding ATL into SITL. Technical report (2009), http://sol.info.unlp.edu.ar/eclipse/atl2sitl.pdf

[9] Akehurst, D.H., Bordbar, B., Evans, M.J., Howells, W.G.J., McDonald-Maier, K.D.: SiTra: Simple Transformations in Java. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 351–364. Springer, Heidelberg (2006)

[10] Sánchez Cuadrado, J., García Molina, J., Menarguez Tortosa, M.: RubyTL: A Practical, Extensible Transformation Language. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 158–172. Springer, Heidelberg (2006)

[11] Barzdins, J., Kalnins, A., Rencis, E., Rikacovs, S.: Model Transformation Languages and Their Implementation by Bootstrapping Method. In: Avron, A., Dershowitz, N., Rabinovich, A. (eds.) Pillars of Computer Science. LNCS, vol. 4800, pp. 130–145. Springer, Heidelberg (2008)

[12] Rensink, A.: Subjects, Models, Languages, Transformations. In: Dagstuhl Seminar Proceedings 04101 (2005), http://drops.dagstuhl.de/opus/volltexte/2005/24

[13] Garcia, M.: Formalization of QVT-Relations: OCL-based static semantics and Alloy-based validation. In: MDSD today, pp. 21–30. Shaker Verlag (2008)

[14] de Lara, J., Guerra, E.: Formal Support for QVT-Relations with Coloured Petri Nets. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 256–270. Springer, Heidelberg (2009)