Automated Model-to-Model Transformation

Foundations of Information Systems Thesis

Radboud University

Stan van Duijnhoven
`s1062589`

Reza Shokrzad
`s1056369`

June 2021

Abstract

This report discusses Data model transformation and automated generation of operations. First we attempt to clarify and summarize the procedure of making automated model transformations and generations in chapter 1. Then, we summarize the material of the lectures from the Foundations of Information Systems course in order to gain a better understanding of the theory and highlight the main topics. The lecture theory is then applied to the automated transformation and generation method and evaluate its applicability. Finally, we propose possible improvements to the theory based on encountered weaknesses.

Chapter 1

Summary of article

1.1 Introduction

Model-Driven Development (MDD) and Model Driven Architecture (MDA) are development methodologies that utilize graphic models [1]. The advantage of such models is the increased speed and ease at which different people with different technical backgrounds involved in the development can communicate abstract concepts to each other. Currently, MDD and MDA model tend to use the initial conceptual schema to derive the final implementation of the system. This final implementation is derived automatically by applying model-to-model transformations. This automatic transformation introduces a problem where the designers of the Conceptual Schema must design a tediously detailed specification of every operation in the system, and the effects these operations have. These operation definitions can be graphically modeled using languages such as UML, ER, and ORM. The UML modeling language is used as the standard notation for the graphic drawing of Conceptual Schemas in this paper. Manoli et al. [1] propose a method for UML class diagram modeling that automates the generation of basic behavior's specification. The aim of this automation is to assist designers in multiple different ways. The most important advantages for designers by using UML transformation are as follows:

- Saving time
- Improving quality
- Reducing errors

1.2 Method

The input of the transformation system takes a UML class diagram. The static aspects of the diagram have to be specified but need no further information beyond this. The method returns a class diagram with extended classes that contain the operations listing the operations of the system state. The method is able to deduce these operations by identifying the structure of the diagram, and its properties and dependencies.

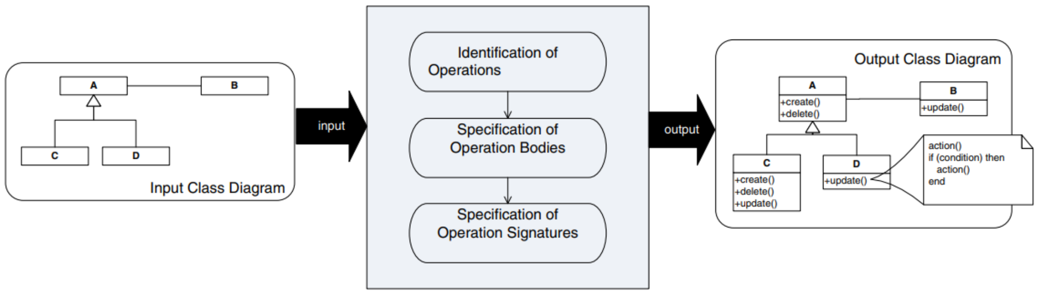

Figure 1.1: Overview of the transformation method

The method proposed by Manoli et al consists of three separate stages of operation. An overview of these stages is visualized in Figure 1.1 The method begins by Identifying Operations as a first step. In this phase, the necessary operations for modifications on values and populations of class elements are identified. Secondly, the Operation bodies are specified. To be more specific, each operation's body, including operation logic and functionality, would be checked to ensure that the structural properties are not violated by the execution. Thirdly, the Operation signatures are specified by deriving them from each action in the operation body. The final signatures of each operation are defined in this final step, only when all dependencies between actions are satisfied. Finally, the authors note that the above method assumes that the defined UML CS is consistent (strongly falsifiable). Each step in this method is further explained in more detail in the following subsections.

1.2.1 Identifying operations

The purpose of the operation identification step of the method is to determine the system operations necessary for modifications on the population and values of class diagram elements. The proper identification of these operations relies on a set of rules, in order to provide basic insert, update and delete functionality. A specific rule has been associated with each type of class diagram element for this purpose.

Classes rule: For each class in the input model, if it is not an Abstract class or a superclass of a covering generalization, generate an operation in the class. If the class does not participate in a restrictive association, generate a Delete operation in the class.

This rule ensures that operations are only created for concrete classes and not abstract classes. This is visible in the above 1.2, note that the abstract class Document does not receive operations.

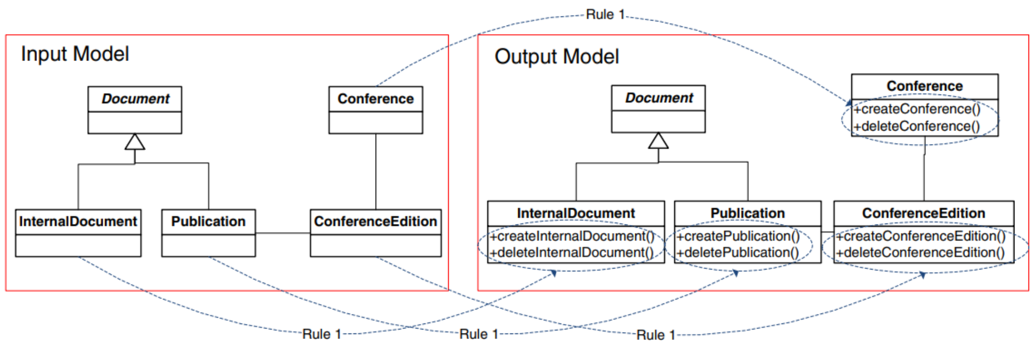

Figure 1.2: Example of the Classes rule in practice

Attributes rule: For each attribute in a class in the input model, generate an Update operation only if the class owns the attribute and is not derived or read-only.

Derived and read-only attributes should not be generated for this purpose because these attributes should not be modified by the designer.

Associations rule: For each association that is not derived, and a participating class can navigate to the class at the opposite end (navigability=true), generate operations that create, delete and update in the class.

Generalizations rule: For each generalization set, generate an Update operation in all subclasses of that class. If the relation is not covering or not disjoint, generate Specialize and Generalize operations.

Operations for moving instances from subclasses of a generalization set to the rest of the subclasses should be generated by this rule. However, specializing and generalizing instances should only be generated by the method to generalization sets that are not covering or not disjoint. Such operations are not needed for generalization sets that have those properties. 1.3 shows an example of this Generalizations rule in practice, creating 2 new operations +updateInternalDocumentPublication() and +updatePublicactionInternalDocument(). The generalization set of the example is covering and disjoint, so it adheres to the Generalizations rule resulting in the creation of these new operations.

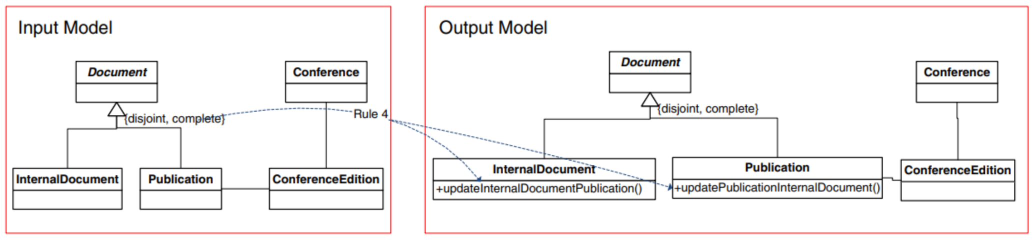

Figure 1.3: Example of the Generalization rule in practice

1.2.2 Specifying operation bodies

The second step in transforming a UML class diagram builds on the first step and consists of generating imperative specification bodies for each generated operation. At first, the operation bodies only contain the basic semantic actions of the operation such as create, delete, update and generalize. These basic operations create a risk that the operation execution does not properly respect the integrity constraints set out and defined in the initial model. This results in the need for additional functionality for operations, in order to satisfy potential dependencies between actions in the operation.

Determining the required actions

The method calculates the actions and preconditions needed for the maintenance of system properties. The method uses two separate steps to determine how these actions and preconditions are computed

1. First, identify the properties of the elements in a class diagram that could be violated when a certain action type is executed.

2. Second, solve or prevent these inconsistencies. For each property-action pair with problems, the actions or preconditions have to be executed together with the action to avoid the property.

The actions that may violate properties of the class diagram are as follows:

- **Minimum Multiplicity**: Certain actions may violate the minimum multiplicity property when its attached value is > 0.

- **Maximum Multiplicity**: Certain actions may violate the maximum multiplicity property when its attached value is $< *$.

- **Delete Propagation**: Certain actions may violate this property, such as when it has links to other items.

- **Changeability**: Certain actions may violate the readOnly and removeOnly constraints.

- **isNotNull**: Certain actions may violate the condition that an action cannot be null.

- **isDisjoint**: Certain actions may violate the condition that an action is not disjoint.
- **isCovering**: Certain actions may violate the condition that an action is not covering

1.2.3 Specifying operation signatures

The purpose of the final step of the UML transformation method to specify the operation signatures of the model. This signature is dependent on the actions that are included in the body of the operation, created in the previous steps. As the number of actions increases, new parameters in the signature are required. This means that every variable parameter in each action also has to be included as a parameter in the operation. Otherwise, the designer of the model would not be able to provide the value of the variable. However, several exceptions must be considered in the method to prevent errors.

- Object variables for actions are not parameters of the operation but are instead created during execution of the operation.
- Parameter variables that already appeared in one of the previous actions, do not generate a new operation parameter.
- The Implicit parameter 'self' is used as replacement for parameters whose type is the class to which the operation is attached.
- Variables for actions that can be obtained by the aforementioned 'self' parameter, are not parameters of the operations.

1.3 Formalization of transformation method

The method described and summarized is defined as a Model-to-Model transformation with a UML class diagram as input. The output of this transformation is a class diagram that contains the set of executable operations. The resulting model provides basic behavior for the developing system. It should be noted that the resulting target model from this method, is a "Platform Independent Model" as described by Model Driven Architecture terminology.

The formalization of this Model to Model transformation is defined in ATL; ATLAS Model Management Architecture [2]. ATL is a hybrid model transformation language, which allows the elements of a source UML class diagram to be transformed into different elements in the output class diagram. The ATL methodology and its modules specify how these elements are mapped from the input model to the elements of the output. The full specification of ATL can be viewed in the ATL paper **input source** and is beyond the scope of this summary.

1.4 Method Evaluation

In the Manoli et al. paper, the usefulness of the automated UML transformation method is evaluated. It is tested for completeness and quality gain compared to manual transformation and specification. Evaluating the completeness is done through a comparison of the operations that were originally specified by the designer with the operations specified by the automatic generation method. The results of this evaluation indicate a high degree of completeness. 69% of operations is completely specified, while most of the remaining 31% is also partially generated. The quality evaluation of the transformation method is performed by analyzing the percentage of missing operations that the method can detect and therefore avoid. The results of the empirical quality evaluations indicate that the method provides benefits to the quality of a system. The additional information that the method generates in the model, can help reduce the number of errors made during the implementation phase. Another benefit is the time saved in development by using an automatic method instead of manual development. The time-saving aspect of the method suggests usefulness for expert programmers and designers too. Even though they may not benefit from the quality advantage due to a lower number of mistakes made, they may benefit from the saved time.

Chapter 2

Foundation of Information System

Information System (IS) is a particular reference to information and the complementary networks of computer hardware and software. So individuals and organizations use IS to gather, clean, process, and distribute data [3]. Object-Role conceptual Modeling (ORM) is applied to simulate the universe of discourse that is a set of entities. Also, ORM's applications are in software engineering and data modeling. Object types are considered fact types (relation types) in an object-role model. Fact types can be of any degree and would grant as object types depending on the technique's strength. These types are called objectified fact types. Figure 2.1 shows a simple graphical representation of a real-world fact type.

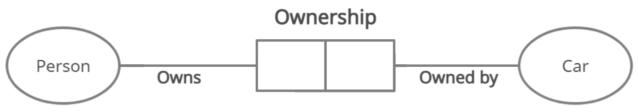

Figure 2.1: a real-world sample of fact types

2.1 Fundamental components

Regarding object-role models, the structure of <$\mathcal{P}$, $\mathcal{O}$, $\mathcal{L}$, $\mathcal{E}$, $\mathcal{F}$, Base, Sub> would define. In this structure, the components respectively stand for predicators, object types, label types, entity types, fact types, base functions, and subtypes. Correspond to this definition, object types (set of $\mathcal{O}$) would be classified as Atomic ($\mathcal{A}$) and Fact types. On the one hand, the label types and entity types are two distinguished categories of Atomic object types with two rules between them $\mathcal{A} = \mathcal{E} \cup \mathcal{L}$ and $\mathcal{E} \cap \mathcal{L} = \mathcal{O}$. On the other hand, fact types are all in Object types that are not Atomic types (F = $\mathcal{O}$ – $\mathcal{A}$). If we define $\mathcal{P}$ as a set of predictors, the function Fact: $\mathcal{P} \rightarrow \mathcal{F}$ would be considered where Fact(p) = f $\Leftrightarrow$ p $\in$ f. It should be mention that all proposed sets are both nonempty and finite. Also, another annotation is named binary relation (Sub $\subseteq \mathcal{E} \times \mathcal{E}$), that $a$Sub$b$ indicates a is a subtype of b or b is a supertype of a.

Figure 2.2 shows a minimal information structure that clarifies the above definitions.

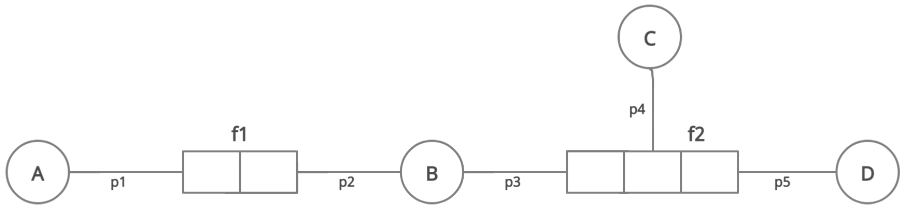

Figure 2.2: an information structure example

According the definitions, Entity types are $\mathcal{E} = \{A, B, C, D\}$, Fact types are $\mathcal{F} = \{f_1,\ f_2\}$, Object types are $\mathcal{O}$ = {A, B, C, D, f_1, f_2}, Predicators are $\mathcal{P} = \{p_1,\ p_2,\ p_3,\ p_4,\ p_5\}$, and we have two sets of $f_1 = \{p_1,\ p_2\}$, $f_2 = \{p_3,\ p_4,\ p_5\}$. As Base: $\mathcal{P} \rightarrow \mathcal{O}$ for each predicator, we would have: Base(p_1)=A, Base(p_2)=B, Base(p_3)=B, Base(p_4)=C, Base(p_5)=D. Since both structures in Figure 2.3 are considered as the same structures, $f_1 = \{p_1,\ p_2\}$ is used instead of $f_1 = (p_1,\ p_2)$ in which the order of elements is important.

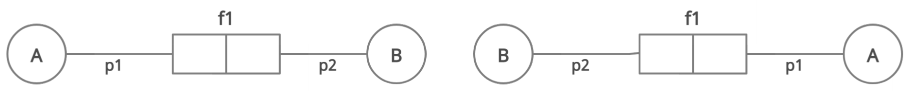

Figure 2.3: The same structures

A structure with two facts f,g would include objectification, if Fact(r)=g, while Base(r)=f. Figure 2.4 depicts the objectification concept with two binary fact types. Thus, if the base of a predicator is a fact type, it is called objectification and a fact type is called objectified, if it occurs as the base of a predicator.

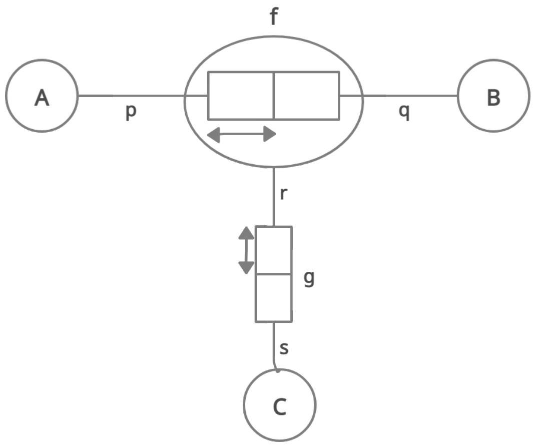

Figure 2.4: An information structure includes objectification

Based on definitions, some other principles should be considered that presented as follow.

- F is a partition of P if and only if:
  - $\forall f \in F[f \subseteq P]$
  - $\forall f, g \in F[f \cap g = \varnothing]$
  - $P = \bigcup\limits_{f \in F} f$
  - $\forall f \in F[f \neq \varnothing]$
- Fact is an auxiliary function that is derivable,
  - $Fact(x) = y \iff x \in y$
- The Universe of Discourse (UoD) is the set of entities over which certain variables of interest in some formal treatment may range [4]. Instantiation (population) is the main part of the UoD that is framed by an

information structure. There is also a mapping from object types ($\mathcal{O}$) to a new set (Ω):

- t: $f \rightarrow \Omega$

To be more detailed, as an example with considering the structure in Figure 2.2 if the population of f_1 includes a_1b_1, a_1b_2, and a_2b_3, there would be the relations below:

- Pop(A)=a_1, a_2
- Pop(B)=b_1, b_2, b_3
- Pop(f_1)=t_1, t_2, t_3
- $t_1, t_2, t_3 : f_1 \rightarrow \Omega \rightarrow t_1(p_1) = a_1, t_1(p_2) = b_1$
- $t_1 = \{(p_1, a_1), (p_2, b_1)\}$ or $t_1 = \{p_1 : a_1, p_2 : b_1\}$

Subtyping (also Subtype) is a binary relation ($\mathbf{Sub} \subseteq \mathcal{E} \times \mathcal{E}$) with this property that each element of $\mathcal{E}$ can be associated with a unique top node. In Figure 2.5 property is Sub asset and House is Sub property, for instance. Furthermore, all subtype relation is represented as arrows and the *pater familias* of object type House is Asset ($\sqcap(House) = Asset$) in which the function $\sqcap : \mathcal{E} \rightarrow \mathcal{E}$ presents *pater familias*.

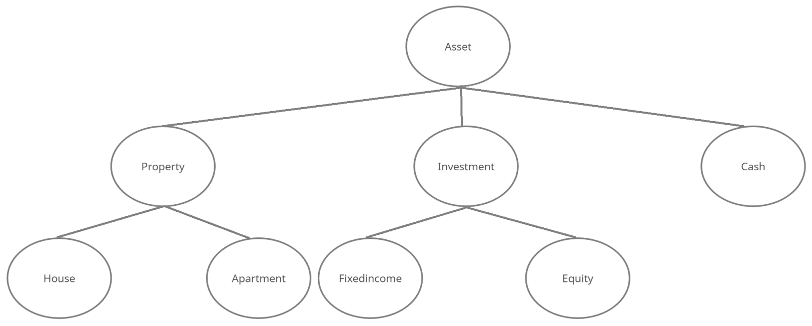

Figure 2.5: An example of subtype hierarchy with top asset

Accordingly, p $\sim$ q denotes two predicators p and q are considered type related, if $\sqcap$(Base(p)) = $\sqcap$(Base(q)). Also, if specific fact types are attached to s but not to t, a subtype s of object type t is used.

2.2 Populations

Universe of Discourse (UoD) is a setting for some part of the fictive world that uses an information structure. Besides, the population or instantiation of the information is the *state* of the UoD. So, **IsPop**($\mathcal{I}$, $\mathbf{Pop}_{\mathcal{I}}$) signifies a population $\mathbf{Pop}_{\mathcal{I}}$ or **Pop** that is a value giving to the object type conforming to the structure as prescribed in $\mathcal{P}$, $\mathcal{F}$ and **Base**, respecting the subtype hierarchy **Sub**.

Some fundamental rules need to be satisfied in defining a population:

- The population of an atomic object type is just a set of values.
- The population of a label type comes from the corresponding concrete domain.
- the population of an entity type comes from abstract domain-containing unstructured values.
- The population of a fact type is a set of tuples. (t: $f \to \Omega$)

2.3 Integrity constraints

In an information structure, some populations are invalid that are called static integrity constraints. Thus, a schema as a part of a relational expression is denoted $\Sigma = \langle \mathcal{I}, \mathcal{C} \rangle$ in which $\mathcal{I}$ in an information structure and $\mathcal{C}$ is a constraint(s), then we have:

$$IsPop(\Sigma, Pop) \equiv IsPop(\mathcal{I}, Pop) \wedge \forall_{c \in C}[Pop \vDash c]$$

Total role, uniqueness, occurrence frequency, set exclusion, set equality, subset, enumeration, total subtype, and exclusion subtype constraints are different types of constraints. For instance, below *total role* constraint is expressed:

$$\bigcup_{q \in \tau} Pop(Base(q)) = \bigcup_{q \in \tau} Val[\pi_q Fact(q)](Pop)$$

If Pop satisfies the above role, it satisfies total(τ) ($Pop \vDash total(\tau)$).

2.4 Information Structure Representations

2.4.1 Diagram

There are several different styles of diagrams that illustrate in figure 2.6.

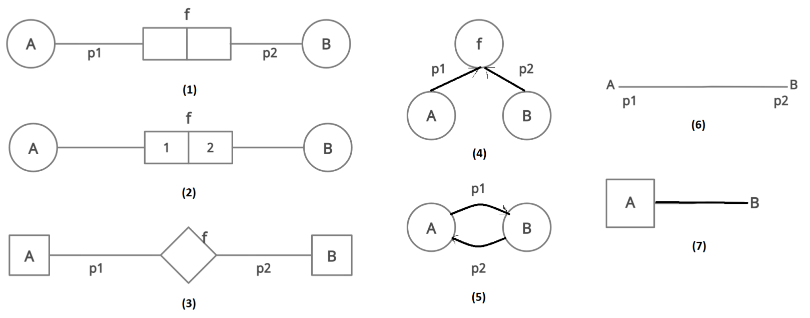

Figure 2.6: Types of information structure diagrams

2.4.2 Tree

There are three schema approaches:

- External (user model).
- Conceptual or PIM (semantic model that focused on the meaning).
- Internal or PSM (physical model that linked to the storage).

Figure 2.7 shows an example of each three approaches. Applying a tree would be helpful to describe a mechanism for the generation of internal representations. The term internal means that users are not aware of the representation's structure, but designers are aware of it.

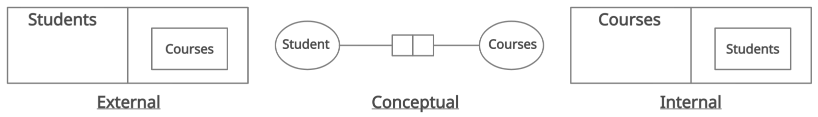

Figure 2.7: Three schema approach

Cutting the connections between certain atomic object types leads to having a forest or set of trees $T = \langle \mathcal{N},\ E,\ l \rangle$ conditionally it would be a labeled graph. Figure 2.8 depicts two examples of transforming from a graph to trees.

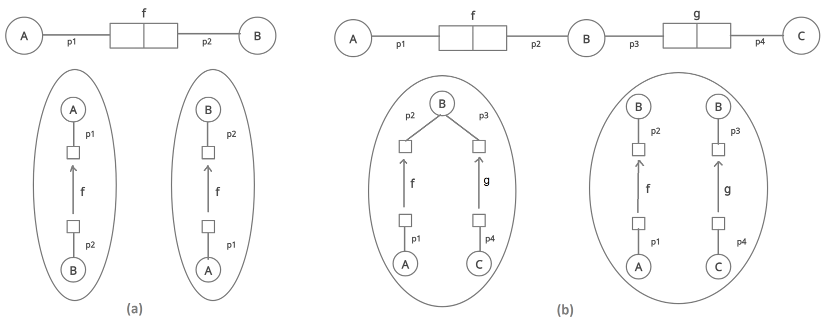

Figure 2.8: (a) two tree derive from the left figure (b) for the right diagram several trees can be depicted that just two of them is shown.

Figure 2.9 displays a bit more complicated information structure and its tree graph. This figure makes the above definitions more clear.

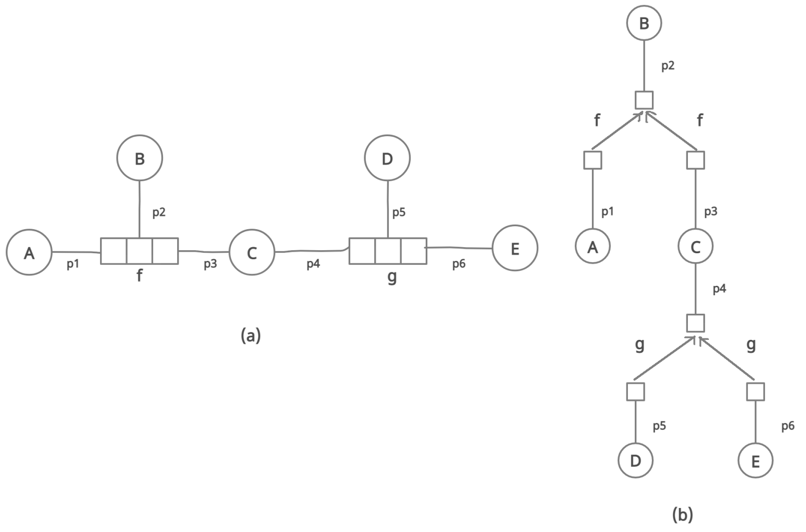

Figure 2.9: (a) a sophisticated information structure (b) the tree diagram of this structure.

In this example, a 3-tuple $T = \langle \mathcal{N},\ E,\ l\rangle$ would represent the tree. In this representation, $\mathcal{N}$ is a set of nodes $\{n_1, n_2, n_3, n_4, n_5\}$, and each n_i is a set shown below.

$$n_1 = \{p_2\}, n_2 = \{p_1\}, n_3 = \{p_3, p_4\}, n_4 = \{p_5\}, n_5 = \{p_6\}$$

In addition, E is a set of edges $\{(n_2, n_1), (n_3, n_1), (n_4, n_3), (n_5, n_3)\}$ and labeling is a function of $l : E \to F$. So, we also have:

$$l(n_2, n_1) = f, l(n_3, n_1) = f, l(n_4, n_3) = g, l(n_5, n_3) = g$$

$$Node(p_2) = n_1,\ Node(p_4) = n_3$$

If a node n is the source of an edge, it would be anchored to this edge by a unique predictor $p \in n$. For instance, in the recent tree, $\mathit{Anchor}(n_3) = n_3 \cap f$. As, $n_3 = \{p_3, p_4\}$, and $f = \{p_1, p_2, p_3\}$, the intersection between them would be p_3.

$$\mathit{Anchor}(n_3) = p_3$$

Additionally, there is a complementary definition named *hook* as a function. A *hook* is a corresponding predicator in destination node n, and it is unique for fact type f.

$$\mathit{Anchor} : N \to P$$
$$\mathit{Hook} : F \to P$$

To be more specific, based on the structure shown in figure 2.10(a), the predicator $|n_1 \cap f|$ is an *anchor* of n_1. Also, the predicator $|n_2 \cap f|$ Is a *hook* of f. If $|f| = 2$ is assumed, figure 2.10 (b) shows the hook and anchor for it.

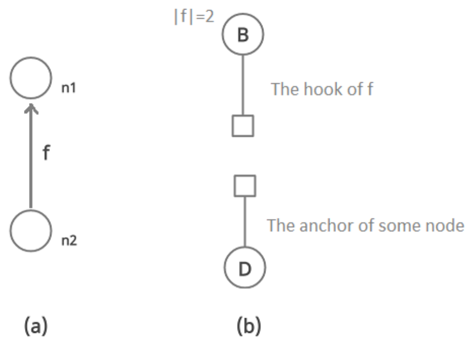

Figure 2.10: (a) an example to understand *hook* and *anchor* (b) an example of those two in a structure with $|f| = 2$

Tree representation provides a lossless transformation that helps not to lose any information. Therefore, the original population of fact types can be produced. Due to the mentioned attribute, the tree representation shown in figure 2.11 (b), (c) would be called lossless or equivalent.

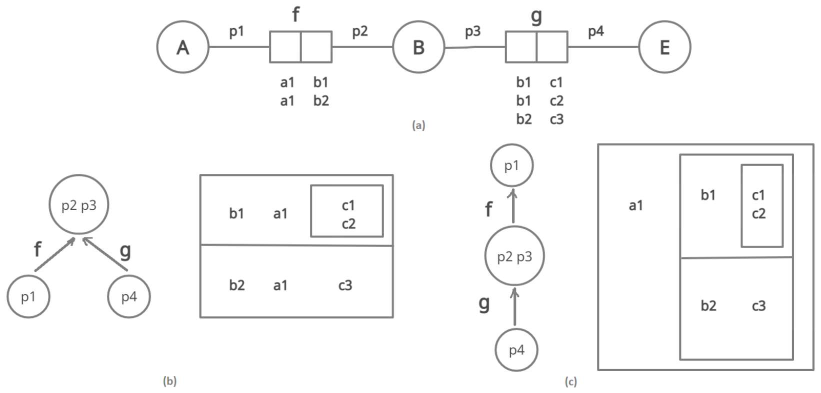

Figure 2.11: (a) An example for Information Structure (b) , (c) Two lossless tree representation of (a)

Table 2.1 is a come up for some definitions so far is discussed. In this term, elements in the first column are following functions. Fact: $P \rightarrow F$, Node: $P \rightarrow N$, Hook: $F \rightarrow P$, Anchor: $N \rightarrow P$, and $l : \mathcal{E} \rightarrow F$.

	Total	Subjective	Injective
Fact	$+$	$+$	$\lvert F\rvert = \lvert P\rvert$
Node	$+$	$+$	$\lvert N\rvert = \lvert P\rvert$
Hook	$+$	$-$	$+$
Anchor	$-$	$-$	$+$
l	$+$	$+$	$\lvert \mathcal{E}\rvert = \lvert F\rvert$

Table 2.1: Experimental results

A significant application of tree representation is being a helpful intermediate language to reduce the contacts of transition between two specific languages. To be detailed, it needs to depict $n \times m$ connections, to move from n possibilities of language 1 to m possibilities of language 2, without any medium (figure 2.12 (a)). While the tree representation would reduce the number of contacts to only n links in front ends and m links for back ends (figure 2.12 (b)).

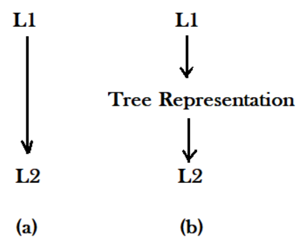

Figure 2.12: Moving from one language to another one (a) without any intermediate (b) with tree representation

Figure 2.13 exhibits an example of the languages' framework with the presence of tree structure.

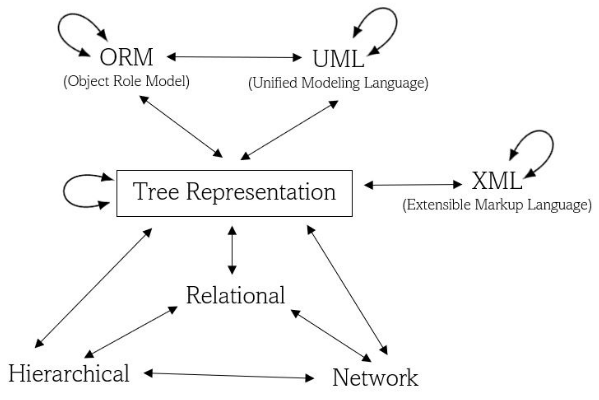

Figure 2.13: The framework of languages

furthermore, it is necessary to indicate how a tree's attributes can calculate from an information structure. There are two provable formulas to compute the number of edges (equation 2.1) and a range of the number of nodes in a tree (equation 2.2).

$$|E| = |P| - |F| \tag{2.1}$$

$$|P| - |F| + 1 \leq |N| \leq |P| \tag{2.2}$$

If all possible tree representations are considered a forest, algorithm 1 is a procedure as a pseudo code to generate the forest.

Algorithm 1: Procedure of Generating a Forest

```
PROC GenerateForest (IN I, OUT T);
$N = \{\{p\}|p \in P\}$;
$E = \emptyset$;
$l = \emptyset$;
# $U$ stands for unused nodes;
$U = N$;
while Can Extend(m, n) do
    Process Fact Type(m, n);
end
$T = (N, E, l)$;
end PROC
```

In each step of this process, nodes n and m would join each other, based on two simple rules.

- if $\{m \in U, n \in N - U\}$, then an existing tree would be extended.
- if $\{m \in U, n = m\}$, a new tree would be created.

Figure 2.14 shows an abstract example of how a forest would be evolving according to algorithm 1.

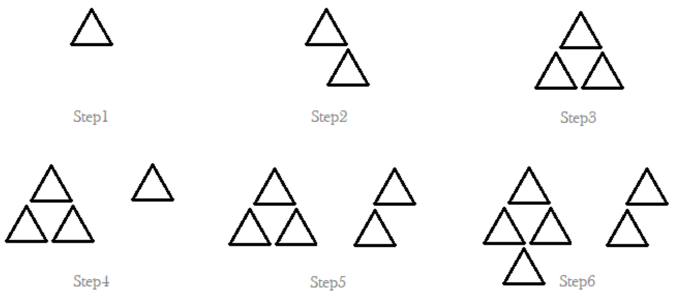

Figure 2.14: An abstract example of a forest developing

As it is mentioned, for each tree, there is a nested table. In this regard, ϕ is a translation of a tree that is recursively defined by traveling the tree from the root downwards. This recursion is ended when all leaves would be attained (figure 2.15).

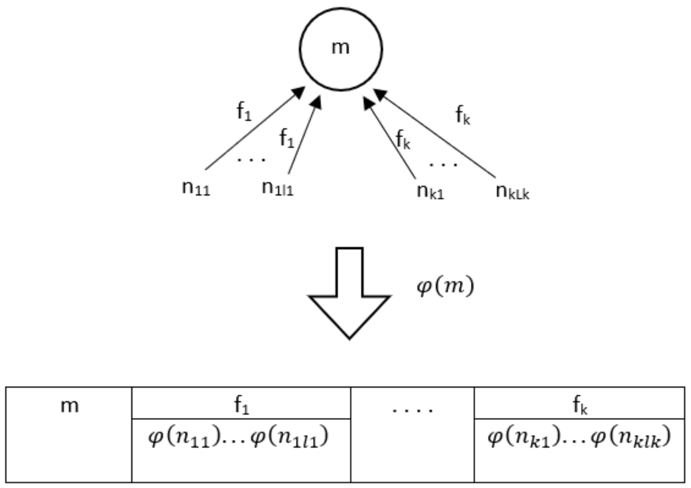

Figure 2.15: An abstract example of a forest developing

Nesting, redundancy (optionals), and preferences concerning the size or depth of the tree are some examples of parameters in the generation algorithm. Regarding redundancy, happening is when there is a one-factor type below another one. For each information structure, there are several possible tree representations that some of which include redundancy. The generation algorithm tries to add $Hook(f)$ to node n if the below statements would be satisfied:

- NoRedundancy = Anchor(n) must be unique.
- NoLossOfInfo = Anchor(n) must be total.
- NoNesting = Hook(f) must be unique
- NoOptionals = Hook(f) must be total.

In addition to these control parameters, there are DepthPreference, and SizePreference as two other examples.

2.5 Transformation

This section discusses the transformation of populations, operations, and structures in some part and transformation of trees, separately. Two distinctive levels of abstraction are considered for databases. These are the conceptual and internal levels that the former concentrates on semantically correct specification and the latter, the internal level, focuses on efficient realization. So, transformation is a stage between these two levels. Populations, operations, as well as structures would be considered for each level in the definition of databases (figure 2.16).

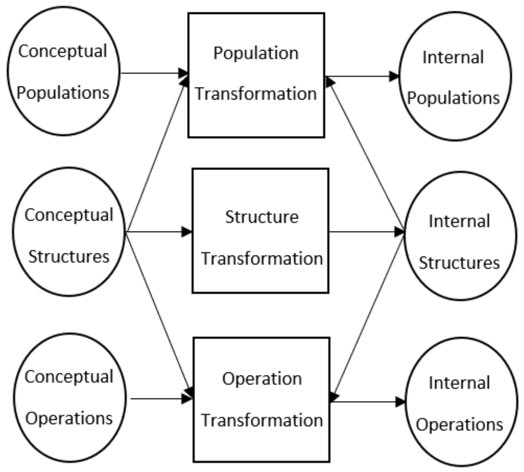

Figure 2.16: Transformation of structure, populations and operations. Boxes represent activities and circles indicate inputs or outputs.

In the other perspective, the population transformation can link tree representation and tree population. Figure 2.17(a) shows the dynamism of the mentioned connection. Founded on this matter, we have both tree (T) and tree population ($Tpop$) representation for each information structure (figure 2.17 (b)).

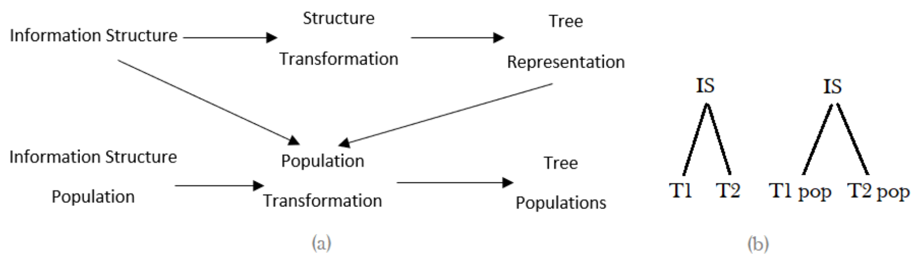

Figure 2.17: Transformation of structure, populations and operations

Based on this assumption, there is an equation (equation 2.3) for each query.

$$\forall q[InfoStructurePop(q) = T_1pop(q) = T_2pop(q)] \tag{2.3}$$

2.5.1 Transformation Rules

1. The partitioning rule

$$\forall_{t_1,t_2 \in Pop(m)}[t_1(m) = t_2(m) \Rightarrow t_1 = t_2] \tag{2.4}$$

2. The fitting rule

$$t \in Pop(m) : (t_1, ..., t_l) \in t(n_1, ..., n_l) \Rightarrow \forall_{1 \le i \le l}[t_i \in Pop(n_i)] \tag{2.5}$$

3. The base representation rule

$$Pop(Base(m)) = \{t(m)|t \in pop(m)\} \tag{2.6}$$

4. The fact representation rule

$$(t_1, ..., t_l) \in t(n_1, ..., n_l) \Leftrightarrow (t(m), t_1(m), ..., t_l(n_l)) \in Pop(f) \tag{2.7}$$

2.5.2 Transformation of database operations

If T is considered as a population set of tuples and t is a tuple in T, the insertion, deletion, modification of population would be the operations that are defined in the following.

1. Insertion of populations

$$Insert(T, t) = T \cup \{t\} \tag{2.8}$$

2. Deletion of populations

$$Deletion(T, t) = T - \{t\} \tag{2.9}$$

3. Modification of populations

   To define the modification operation, it needs to assume a function m with domain $Dom(m) \subseteq Dom(t)$ that for modifier m, the *complementary domain* is defined by $ComDom(m) = Dom(t) - Dom(m)$. Besides, the modified tuples will be described by equation 2.10. So, modify operation can be characterized by equation 2.11.

$$M(t, m) = m \cup t[ComDom(m)] \tag{2.10}$$

$$Modify(T, t, m) = T - \{t\} - \{M(t, m)\} \tag{2.11}$$

2.6 Solution spaces

This section covers solution spaces, their walkthroughs and their theoretical consequences. Solution space S is the set of all (theoretically) possible tree representations for a corresponding conceptual model. Solution space S consists of $|f|$ number of hyperplanes. Hyperplanes can be defined as geometric shapes within four or more dimensions. The number of joins within a hyperplane can be described as follows:

Hyperplane $\mathcal{V}_i \underline{C}\ S$ contains i joins.

Where i is greater than or equal to 0 but greater than or equal to $|f| - 1$. The solution space can be derived from a set of representations. An example of one way of displaying a solution space mechanism is visible in Figure 2.18.

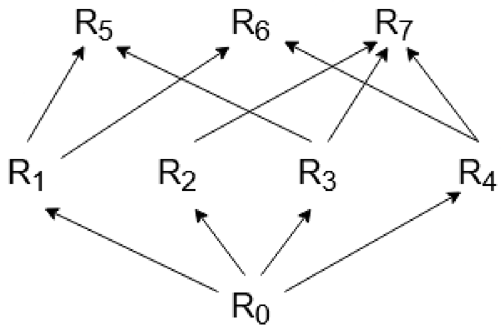

Figure 2.18: Solution space example

These abstract solution spaces make it easier to verify statements about the solution space. The connecting lines in the above model indicate whether a representation R can be transformed from another representation. This is true even if the representation is indirectly transformable.

- Statement $J(\mathcal{R}_0, \mathcal{R}_1)$ is **true** because $\mathcal{R}_1$ can be transformed from $\mathcal{R}_0$.
- Statement $J(\mathcal{R}_1, \mathcal{R}_2)$ is **false** because $\mathcal{R}_2$ can be transformed from $\mathcal{R}_1$.
- Statement $J(\mathcal{R}_0, \mathcal{R}_5)$ is **true** because $\mathcal{R}_5$ can be transformed from $\mathcal{R}_0$ **via** $\mathcal{R}_1$ or $\mathcal{R}_3$.

This results in several theoretical consequences to consider.

- The statement $J(x, x)$ is always true. This is a reflexive statement, the property holds between an element and itself.
- The statement $J(x, y) \wedge J(y, z) \Rightarrow J(x, z)$ is always true. This is a transitive statement.
- The statement $J(x, y) \wedge J(y, x)$ means that $x = y$. This is is an anti-symmetric statement (Note: different from *asymmetric*).

Because J is reflexive, transitive and anti-symmetric, it is classified as a **partially ordered set**.

2.7 Tree representation encoding

Instead of drawing a tree representation, an encoding can be used. Encoding is a powerful mechanism that shortens the tree representation to a code without drawing it. For encoding, the following rules must be considered:

- The predicators numbered from 1 to $|P|$ (number of predicators).
- The roots are numbered from -1 to $-|R|$ (number of roots)

The reason for the negative number assignment to the roots is to be able to distinguish roots from predicators. The encoding is a function a, where each predicator is encoded by a value either a predicator or a root.

$$\text{Encoding } a : P \to P \cup R$$

The encoding works through the following steps. If P is in a root, take the root number. If P is not in a root, take the anchor of the node.

Written as a a function: $a(p) =$

$$\begin{cases} \text{Node(p) if Node(p)}\leftarrow\text{R} \\ \text{Anchor(Node(p) if Node(p)! }\leftarrow\text{R} \end{cases}$$

Chapter 3

Automated Transformation

3.1 Introduction

In chapter 1 we have summarized the paper detailing a methodology developed by Manoli et al. that automates the transformation of UML Class Diagrams in a Model-to-model (M2M) approach. In chapter 2 we have addressed material from the lectures of the Foundations of Information Systems course. The lecture material covers topics regarding transformation of models and represenatations.

- Transformation of conceptual models into tree representations.
- Transformation of populations
- Transformation of tree representations into an encoding.
- Transformation of database operations

As Manoli et al. note, the quality of the transformed models is a concern because of possible human errors during transformation. The time spent on manual transformation is also a drawback, especially for novice designers such as students. The method described in the Manoli paper could be adapted to include the different kinds of transformations mentioned. If the method were to be successfully adapted, it could offer improved quality and time saving when transforming models.

3.2 Adapting the transformation method

We consider the method by Manoli et al again where the 3 steps are laid out that are required to transform an Input Class diagram to and Output Class diagram with generated Operations. These steps have been detailed in Chapter 1.

1. Identify Operations
2. Specify Operation Bodies
3. Specify Operation Signatures

For the execution the method, a object oriented programming language should be used. The theory described in the lectures is suitable for object-oriented languages because of the different kinds of objects involved, such as Entities, Facts and Predicators. The programming language used in this paper for the example code is Java, one of the most popular Object-oriented languages. It should be noted that this code is pseudo-code, which may not be correct because we have limited programming experience. We can start by creating an Object O:

```
import java.util.ArrayList;
public interface Object {
public String getName();
public ArrayList<String> Pop();
public void setP(Predicate P);
public ArrayList<Predicate> getP();
public Type getType();
```

3.3 Automated Populations Transformation

As it is indicated in section 2.3, in the class diagram all distinct elements have a specific type of values or population. Based on the three-step method of the paper, it is necessary to identify which operations should be defined to achieve the modifications on the population of those elements in the class diagram. This step needs to have a determination of the functionalities on the population such as insert, delete, and update to avoid automatically generating unnecessary operations. Due to what the paper says, in section

1.2.1, four associated rules are corresponding to each element of the class diagram (class, attribute, association, and generalization set). These conform with what rules have been discussed in the lecture note, in section 2.6.

According to the 4 rules of the automation process which is articulately reported in Manoli et al work, the creation, deletion, and modification of each activity like the transformation of the population can be easily implemented automatically, with some simple steps. For instance, in accordance with figure 2.16, if we consider instances of each input (boxes) of population transformation, such as instances for conceptual populations, conceptual structures, or internal structures, we would fulfill the mentioned rules.

In actual fact, conceptual and internal levels are information structure $\mathcal{I}$ and tree representation T, respectively. Thus, we have two types of making populated a structure that is explained followingly.

3.3.1 Information Structure Population

Along with Conformity Rule, we can design an automated system to make populated an information structure ($\mathcal{I}$). As it is elaborated in section 2, the population of each type is clear. For instance, we determined the population of an atomic object type is a set of values, the population of a composed object type (fact type) is a set of tuples, and a tuple t of a fact type f is a mapping of all its predicators to values of the appropriate type. So, making a $\mathcal{I}$ populated automatically would be feasible in a straightforward way.

3.3.2 Tree Population

Where the population of a nested relation like nested tuples is the approach that tree representations are populated. Also, we state such tuples as nodes in the forest. Due to Partitioning Rule (equation 2.4), we can easily initialize the population of a tree. To be more detailed, if we denote t as a tuple and m as a node it is possible assigning each tuple unique value to a m.

3.3.3 Transformation of these Populations

In this section, we are going to generate a population of tree T from a population of information structure $\mathcal{I}$. If m would be a node in T each instance of the corresponding base a unique tuple will result in Pop(m), then we have base representation rule, $Pop(Base(m)) = \{t(m) | t \in Pop(m)\}$. Regarding the children of this node, $Ext(f) = n_1, ..., n_l$ would be the children in which f is fact type with $Hook(f) \in m$. In the mention transition each tuple $t \in Pop(m)$ the assignment $t(n_1, ..., n_l)$ is defined by:

$$if \langle t(m), t_1(n_1), ..., t_l(n_l)\rangle \in Pop(f) : \{\langle t_1, ..., t_l\rangle \in t(n_1, ..., n_l)\} \qquad (3.1)$$

This operation is totally compatible with the three-step rule in Manoli et al work.

3.4 Tree representation to Encoding

When looking at a tree representation, such as the example in Figure 2.18 in section 2.7, we can see that the Representations R, numbered from 0 to $|R|$. The possible transformations between different Representation R's is indicated by a directional arrow. The resulting encoding is a more efficient and shorter way of representing a tree. This encoding can also be reverted to a tree representation through the process of decoding. In order to transform a tree representation to an encoding where the input model is a tree representation and the output is an encoding table, we propose a three-step method. Each step is defined in the following section, with a schematic overview including examples from the lecture in Figure 3.1.

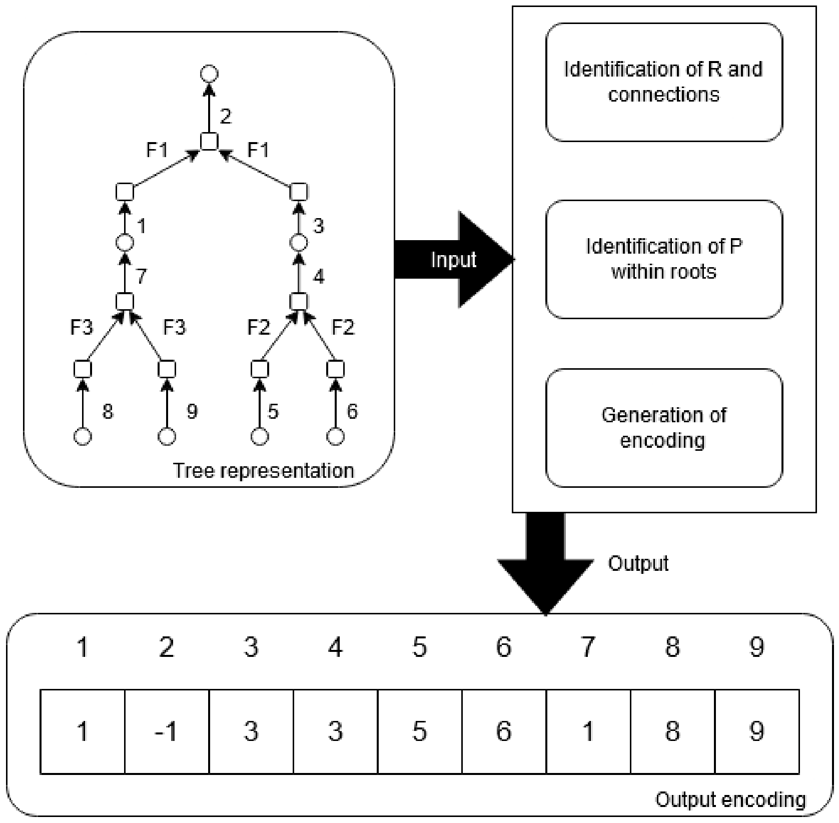

Figure 3.1: a

3.4.1 Identification of Roots and their connections

First, the Roots and their connections are identified. This is important for the third step in the method where the encoding is generated. In order to distinguish the predictors from the roots, and for the method to recognize them as such, the Roots R must be numbered as negative values. If any of the Roots have positive values, the method should take the negative value of the given root.

3.4.2 Identification of Predicators inside of the root

Second, the Predicators that are inside the root are identified. This step is required for determining what value should be assigned to it in the encoding (step 3). For each Predicator, check if:

$$Node(p) \leftarrow R$$

3.4.3 Generation of the encoding

Now that the roots, connections and predicators are determined, and we know which Predicators are inside the root, we can start the encoding.

```
if (Node(p) <- R) {
   P = Node(P)
   } else {
       P = Anchor(Node(P))
   }
```

The above piece of code attempts to assign the predicator P as the root number if P is inside the root. Else (meaning if P is not inside the root), assign the anchor of the node. The resulting encoding from the execution of this method should look like Output encoding from Figure 3.1, where the numbered predicators are linked with a value determined by the method.

Chapter 4

Evaluation of Applicability

In this chapter, we discuss the applicability and suitability of the theory of the lecture material. We start by describing the strong parts of the theory, specifically when applying it to the Universe of Discourse that we used. We then describe the weaknesses of the theory in the applicability to our automated transformation application.

4.1 Strengths of the theory

In general, making transformations automated would reduce human errors in large-scale structures. In this way, designers would face fewer challenges due to bugs. As a result, the transformation process would be more reasonable and time-saving and increase the robustness of the process. Apart from this, straightforward implementation, owing to the rigidity of theories, is another strength point of automation.

4.1.1 Human error reduction

In consequence of the evaluation phase of the paper, raised in section 1.4, the number of satisfied constraints and generated methods with the proposed method is more than designers' works. Although we could not evaluating our transformation approach, we can accept that the result for large-scale systems would be better based on the proposed technique.

4.1.2 Rigidity of the theory

A clear strength and convenient property of the theory is that it is very rigid. The processes in the theory, such as the transformations, are well-structured and well-defined. This is expected for field of study that has been around for a long time, with many different modelling methods and theory developed and improved over time.

4.2 Weaknesses of the theory

We encountered some problems during applying the theory to the Automation method described in the Manoli et al. [1].

4.2.1 Retrieving information from models

The main difficulty we encountered is that some models, such as a tree representation, are difficult to adapt to the point where an automated method could recognize them. In order to transform models, the method must first retrieve essential information from the model including the different R representations and connections to other R's. This made it difficult to create code that can interpret drawn models, which is why the code suggested in this project for the automation is limited to pseudo-code that cannot be executed.

4.2.2 Costly Debugging

Like any other mechanical process, the recommended method has two cost-risky sides. While the method helps systems to be independent of designers, there is a drawback related to the maintenance of programmed results. Sometimes, debugging an extensive system can be more high-priced than using transformation by a designer.

Chapter 5

Improvement Propositions

In this chapter we suggest improvements to the theory that may make the theory more adaptable to new methods, primarily computer-executed functions.

5.1 Adaptability to automation

As described in Chapter 4, some models from the theory such as tree representations and populations may be difficult to automate. Because automation brings possible quality improvements and time saving for designers of models, it may be beneficial to make the theory more suitable for automation. To this end, we propose the following suggestions for improvement of the theory

5.1.1 Encoded models

Instead of focusing on visual models such as tree representations, walk-throughs and other visual models, we suggest instead to focus on encoded models. Although visual, drawn models are generally more suitable for understanding by humans, they may prove difficult for parsing by computers. Encoded models such as the tree representation encoding, are easily understood by automated methods because they consist mostly of numbers instead of visual elements which may be more open to interpretation and error. Now that computers become more and more entrenched into our daily lives, they could also increasingly assist us in designing and transforming models. For these reasons, using encoded models over drawn versions appears as a solution.

5.2 Semi-supervised Automation

To solve the problem mentioned in section 4.2.2, constructing a semi-supervised system of automated transformation would be effective. To implement this strategy, we can consider two probable methods. For one of them, it is required to detect which 3-step rule is violating more. As our solution is semi-supervised, a designer can regulate the performance of the method in the part with more deviation. In the other one, we can focus on the violation of the tasks. For example, at first, we detect which task among the population, operation, or tree representation transformation are more likely would face bugs. Then, we can assign the designer to conduct this part.

Bibliography

[1] Albert, Manoli, et al. "Generating operation specifications from UML class diagrams: A model transformation approach." Data Knowledge Engineering 70.4 (2011): 365-389.

[2] BÉZIVIN, Jean; JOUAULT, Frédéric; TOUZET, David. An introduction to the atlas model management architecture. Rapport de recherche, 2005, 5: 10-49.

[3] `https://en.wikipedia.org/wiki/Information_system#cite_note-1`

[4] `https://en.wikipedia.org/wiki/Domain_of_discourse`