Introduction to the Graph-Oriented Programming Paradigm

Olivier Rey[1][0000−0003−4462−3712]

GraphApps
rey.olivier@gmail.com – orey.github.io/papers

Abstract. Graph-oriented programming is a new programming paradigm that defines a graph-oriented way to build enterprise software, using directed attributed graph databases on the backend side.
Graph-oriented programming is inspired by object-oriented programming, functional programming, design by contract, rule-based programming and semantic web. However, it proposes a consistent approach and enables to build enterprise software that does not generate technical debt. Its use is particularly adapted for enterprise software that must manage high complexity of data structures, evolving regulations and/or high numbers of business rules.
In this article, we present a first high level overview of the paradigm and how this new programming model can solve the two core issues of the technical debt, structural and temporal couplings.

Keywords: graph · design · software architecture · software engineering · programming.

1 Maintenance and Evolutions in Enterprise Software

The way the software industry currently builds the enterprise software generates a lot of "couplings": inside the code, inside the data and between the code and the data. In this section, we propose to show that those couplings are intrinsically attached to the way the software industry is building software and not to the semantics of the business itself. The paradigm we will study is object-oriented programming and relational database persistence.

1.1 Structural Couplings in Enterprise Applications

In order to represent what we intend by couplings, we define three levels of thinking: the *semantic level*, often used in the functional analysis step [8], the *code level*, represented by UML diagrams, and the *database level*.

In Fig. 1-A, we focus on the case where A has a `(0..1)` relationship with B. We can note the semantic concepts $\overline{A}$ and $\overline{B}$ and this dependency as $\overline{A} \rightarrow \overline{B}$. At the code level, the `Class A` is having an attribute `myB` of type `Class B`. Generally a method of `Class A`, `methodA1`, will invoke on `myB` a method of `Class B`, here `methodB1`. In this kind of simple aggregation, `Class A` owns two kinds

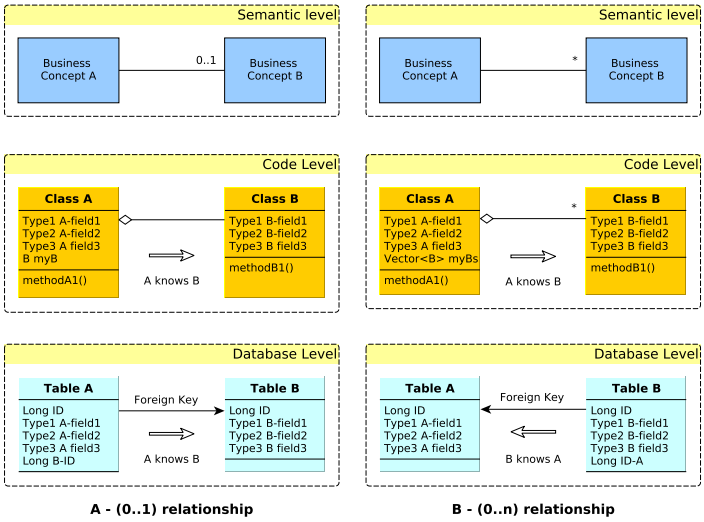

Fig. 1. Couplings in the `(0..1)` and in the `(0..n)` relationships

of knowledge about `Class B`: how to navigate from an instance of `Class A` to and instance of `Class B` (topology knowledge), and the prototype of `methodB1`. Inside the database, this coupling will be in the same direction as in the code: the `Table A` will contain a foreign key to `Table B`. Both in the code and in the database, we have the relationship included inside `A`. We will name C_c the code coupling, and C_d the database coupling.

The overall couplings in terms of code and database can be written the following way: $A(C_c, C_d) \rightarrow B$.

In Fig. 1-B, we are analyzing the `(0..n)` relationship, that we will note $\overline{A} \rightrightarrows \overline{B}$. At the code level, `myB`, attribute of `Class A`, will be a list or array of some sort of instances of `Class B`. At the database level, `Table B` will have a foreign key on `Table A`. This leads to: $A(C_c) \leftrightarrow B(C_d)$[1].

Enterprise software manipulating dozens to hundreds of business concepts, each relationship between them generating couplings, they are very hard to modify and evolve, this from the very beginning of their construction. Each modification of the semantic model, for instance going from $\overline{A} \rightarrow \overline{B}$ to $\overline{A} \rightarrow \overline{C} \rightarrow \overline{B}$, or from $\overline{A} \rightrightarrows \overline{B}$ to $\overline{A} \rightrightarrows \overline{C} \rightarrow \overline{B}$ requires a lot of software tasks that are not related to the semantics of the business but more to the way the software is built.

The structural couplings in enterprise software are a big part of the technical debt. The technical debt [3,10,6] is generally defined in terms of costs: it is the

[1] We are studying here the two simplest cases. In real life software, we can have more complex class to table relationships, one class being serialized in several tables or several classes in the same table. But, in terms of cardinality, we end up creating the same couplings.

difference of costs between the cost of the implementation of a software module `Y`, developed alone, and the costs of the same module `Y` developed in the context of an evolution of an existing system `X`. This enables to compare the function point [14] cost in both situation.

We can say that our OOP/RDBMS approach of the semantic concepts and relationships implementation is *not optimal*, because it adds technical couplings to the business semantics.

1.2 Temporal Couplings in Enterprise Software

Many changes in the applications come from changes in requirements, regulation or laws. Those changes are, most often, time-based changes. In a lot of enterprise software, many structural changes are time-dated and contextualize business rules with time. In most cases, old data must be governed by old business rules and new data (possibly with new structures) by new business rules.

We show in Fig. 2 a sample of the impact of a regulatory change in a software, which version is going from `Vn` to `Vn+1` between `Tn` and `Tn+1` (semantic view). We will consider the modification of `P2` and `P3` programs that are respectively embedding business rules `BR2` and `BR3` (analytic view). We will also suppose that the data in `Vn` are stored in data structures in version `Vk`.

For the evolution from `Vn` to `Vn+1`, we will perform the following tasks:

- The database will be updated from version `Vk` to version `Vk+1`;
- The business rules will be upgraded from version `M` to `M+1` and `L` to `L+1`;
- We will test if the new business rules `BR2 M+1` and `BR3 L+1` correctly apply to newly created data in version `Vn+1` stored in the version `Vk+1` of the data structures;
- We will test that the old business rules `BR2 M` and `BR3 L` still apply to old data in version `Vn` recently migrated to new data structures `Vk+1` (non-regression testing).

This evolution creates what we can call a "temporal coupling" (in red in the analytic view): we attach data version `Vn` to data structure version `Vk+1` and we hide two versions of the same business rules in the evolutions of `P2` and `P3`.

This very common practice comes from the fact that we want to consider as being the same entities, objects which structure evolves with time. Indeed the real need for evolution is that version `M+1` of `BR2` is managing data version `Vn+1` on data structures in version `Vk+1` and the same for `BR3`. Concerning the past data, it depends if the business rules that apply on those data are limited to the past or not. Most often, in enterprise software, past data do not need to be migrated because the business rules that apply to them are also time-dependent.

Actually, considering our ways of storing information in RDBMS, it is not easy to version the storage structure[2]. So, when a new requirement implies an evolution of the storage structure, this storage has to become the new storage *for*

[2] In some very big accounting systems, we can find different structures for the same entities depending on the time.

all data of the same kind: old ones, that potentially will never evolve anymore and do not need the structural modification, and new ones that require the structure modification for the new business rules to apply. And, as long as all data (old ones and new ones) are located inside the same table, it is quite normal to have only one version of program to manage them, whatever their age. The result is after dozens of modifications, the enterprise software contains many versions of business rules in the same programs that apply to a single version of data structures hiding many versions of data. The technical debt is huge in those cases.

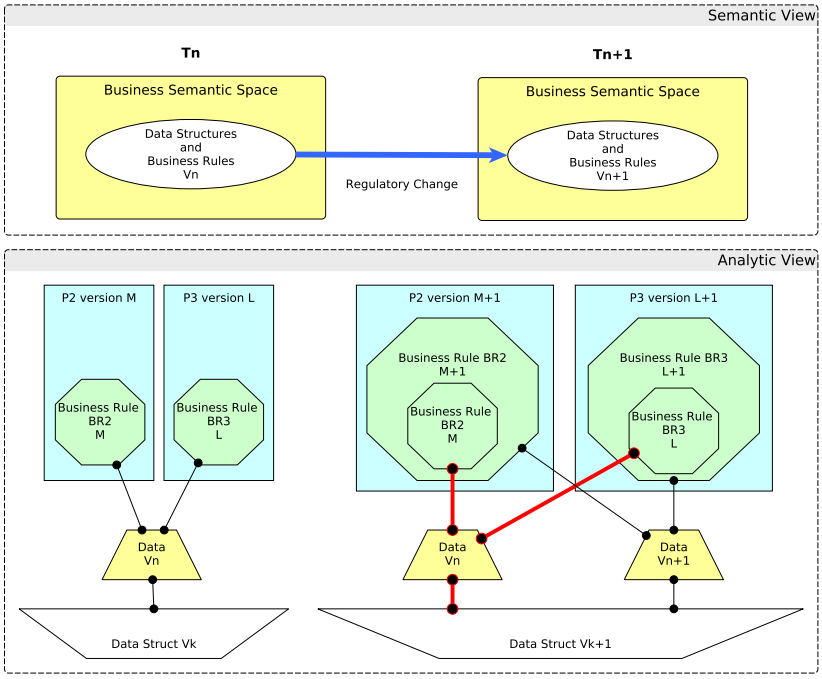

Fig. 2. Evolution management in enterprise software

Conceptually, this phenomenon introduces the notion of domain of data for a business rule, notion on which we will come back in section 3.2. In fact, in the current software industry databases and programming languages, we have almost no time-based constructs, nor any *timelining* software concepts and best practices for software engineering.

1.3 Addressing the Technical Debt

After the initial delivery, the enterprise software scope changes to accommodate new requirements. Depending on the location of the modifications in the soft-

ware, the evolution can be from simple to very hard to implement. For instance, an evolution of the software core system, after some years, will be extremely difficult to do, the impacts being so huge that the cost of such a modification could become comparable with a full redesign and rewriting.

For decades, the software industry found work-arounds to address the technical debt problem, without really questioning the nature of the technical debt itself.

The object-oriented *design patterns* [11], for instance, can be seen as *a set of empiric recipes* proposing, amongst other objectives, to enhance application extensibility. In most IT projects however, the capability of software engineers to properly anticipate, at design time, the directions for software evolutions is quite *low*, with or without design patterns. The reason is simple: the object-oriented design is a methodology that aims at creating the best design (or the less-worst) considering a set of use cases (the scope). If the scope changes, the new use cases can have a deep impact on the original design. Design patterns are propositions to ease future maintenance and software extensibility. But, if they can work in some cases, most of the time, the software evolves in directions that were not anticipated by the original designers[3].

Software architecture [2,1,7] is at the heart of a very large literature and can be considered also a way to address the technical debt. By defining cautiously the architecture of a software, software architects can limit the impact of change, for instance a database change or a web service signature evolution. This discipline is a set of rules to identify big blocks inside the application and link them together. Those methods help to reduce some of the coupling that are more related to software component dependencies and separation of concerns. But, when the core business model needs an evolution, several layers and components, plus the database, still need to be updated.

Refactoring methods [9] are often used when the software must evolve to integrate new requirements, or in the case of code stabilization. Generally a refactoring project will have the 4 following parts: a redesign of at least one part of the software; structural changes at the database level; data migration (from their original structure to the new one); software non-regression testing. Depending on the complexity and the scope of the changes, refactoring projects are generally risky and costly. The refactoring methods are always analyzed by the software industry in comparison with partial or full rewriting projects, that are also very costly and very risky.

Despite all those efforts to minimize the technical debt in enterprise software, during the construction or the maintenance phases, the software industry never solved the technical debt problem. Today, the majority of software engineers work in software maintenance and a huge portion of the IT budgets are spent in overcoming the technical debt.

[3] We can also note that misused design patterns are increasing the technical debt.

2 Introduction to Graph-Oriented Programming

We propose, in this section, to examine the basic structure of information in the graph-oriented programming paradigm, first through modeling concepts then through code requirements.

2.1 Node Types and Relationship Types

Concept types will not be represented anymore by classes, as the object-oriented design, but by node types (yellow circles). Relationships will not be managed anymore by attributes included in the container class but by a specific construct: a relationship type (blue parallelogram). Relationship types have directions and they cannot exist with a source node and a target node. Node and relationship types can have attributes (see Fig. 3-A). The instantiation of those types creates a graph of nodes connected by relationships (attributed directed graph). We will note $A - [R] \rightarrow B$ a synthesis of Fig. 3-A.

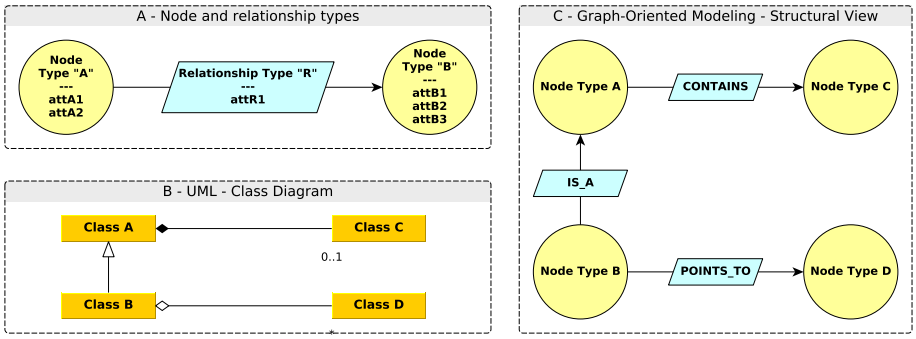

Fig. 3. UML versus graph-oriented modeling

Fig. 3-C is an attempt of translation of the UML model of Fig. 3-B in a graph model. In order to define relationship types, we were bound to name them and so to introduce new semantics, as we can do in semantic modeling [15,5]: `CONTAINS`, `IS_A`, `POINTS_TO`) are new relationship types that we did not have in UML. Because the object-oriented programming languages are working in a certain way, UML considers aggregation, composition and extension as being *structural relationships*, because they structure the code it self. In graph-oriented modeling, we have no distinction of the kind: relationship types can be structural *in their semantics*, i.e. relatively to their meaning. However, there is no necessity to tag relationship types as being structural or not.

Naming relationship types pushes us to answer to new questions. Let's take the sample of a `Class Car` aggregating a `Class Wheel` and a `Class Cocktail`

aggregating a `Class OrangeJuice`. We have several ways of modeling in a semantic/graph way. We could mimic UML structural generic aggregation relationship writing $Car - [contains] \rightarrow Wheel$ and $Cocktail - [Contains] \rightarrow OrangeJuice$. But we could also propose: $Car - [hasPart] \rightarrow Wheel$ and $Cocktail - [hasIngredient] \rightarrow OrangeJuice$. This is a quite common design topic in the semantic web world [5]. The fact of having two relationship types can enable the software designer to think about having different business rules attached to the semantics of those relationships: for instance, the $[hasPart]$ relationship can be deleted and the $Wheel$ can have its own life cycle apart from the Car, whereas the $[hasIngredient]$ relationship could not be removed once an instance of $Cocktail$ was built.

Specialization UML relationship will also be managed by relationship types, which can, as in the semantic web be used at several levels. For instance, let us consider a `Class Animal` and a `Class Dog` specializing it. We can write in a graph-oriented design approach: $Dog - [isA] \rightarrow Animal$. But we can generalize this specialization mechanism to other cases such as: $DougsDog - [instanceOf] \rightarrow Dog$, this being in the same model, $Model - [governedBy] \rightarrow Metamodel$.

All that to say that in a graph-oriented design approach, we can have most of the same modeling issues that we find in RDF/RDFS. The main difference between this modeling approach proposition and the semantic web is that a relationship type *cannot be considered* as a node type, whereas in the semantic web, triples can have as a subject, a predicate of another triple.

Expressing aggregation, composition and specialization with relationship types opens a large number of design possibilities that we do not have in the object-oriented programming world. For instance, we can associate instances to a node that will represent their type, or associate a model with nodes that are representing its metamodel. The consequence is that it becomes easy, in the same model, to manipulate multiple levels of abstractions.

2.2 About Constraints on Relationship Types

We will have sometimes the need to put constraints on the source and/or target node types that are meaningful for a particular relationship type. In case the relationship type accepts all types or as its source or as its target, we will use the convention $*ANY\_TYPE*$ to designate this freedom. This is equivalent to defining the domain and range of a predicate in RDFS. For a relationship type R, we could write in RDFS `R rdfs:domain *ANY_TYPE*` and/or `R rdfs:range *ANY_TYPE*`.

The cardinality constraints are also important to take into consideration. As the relationship types are outside the node types, nothing prevents an instance of node A, `A1`, to have many instances of the same relationship type R, $\mathtt{R}_1, \mathtt{R}_2, \ldots, \mathtt{R}_n$, to the same instance of B, `B1`. R should be able to come with a cardinality oriented constraint like we have in UML.

The crucial fact to notice about relationship types constraints, on a software engineering point of view, is what entity is responsible to check and enforce them.

In object-oriented programming, the class managing the relationship masters the source or target type (type of *this*) and is able to control the cardinality (by defining a fixed-sized array or list as its attribute). In graph-oriented design, nor the node type, nor the relationship type can manage the constraints. It is the role of graph transformations (cf. section 3).

2.3 Introducing the Domain Concept

Node and relationship types can be grouped by semantic space. We propose to name those spaces *domains*, because in the software industry, the notion of business domain is widely used [7] to represent business domains. During the graph-oriented modeling phase, the idea will be to analyze the business and to group node and relationship types by domains. We can imagine a lot of kinds of domains: business, technical, utility, metamodel, etc.

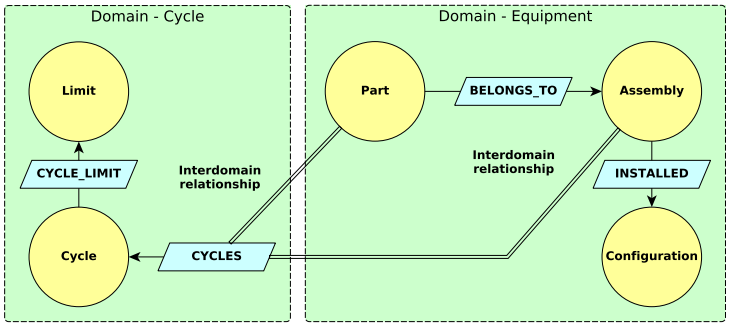

Fig. 4. Domain concept illustration and inter-domain relationships

The Fig. 4 shows a multi-domain approach through a sample from the aerospace maintenance. Some relationship types can be *bridges* or *inter-domain* relationship types (partially represented with a double arrow in Fig. 4). This point is very important because this inter-domain connectivity is one of the crucial sources of software extensibility in graph-oriented programming.

2.4 About Graph-Oriented Modeling

With the simple rules that we have defined, it is possible to model every business domain that is the core of enterprise software. By the use of domains, it is also simple to structure the models in parts and to identify inter-domain relationships, which corresponds to the idea of software module. If we take back the notion of section 1.1, the semantic dependencies $\overline{A} \rightarrow \overline{B}$ and $\overline{A} \rightrightarrows \overline{B}$ are translated in design by $A - [R] \rightarrow B$, A and B being node types and R a relationship type. Our design model does not alter the concepts $\overline{A}$ and $\overline{B}$ and does not add coupling to the semantic representation once in the design phase.

This is a major step ahead, especially if we consider that this modeling is compatible with the way the attributed directed graph databases of the software industry are built. Indeed, in data also, the translation of the semantic dependencies will still be $A-[R] \rightarrow B$. In a way, this evolution in the design approach, compared to object-oriented design, can be considered as *optimal* compared to the semantic dependencies we have to model.

This means that the design process can become much nearer from the way the business people are describing their business, in the context of enterprise software. If they are able to draw the various entities and their relationship, the modeling should be easily done and shared with users. We also saw that we could model instances in that way.

2.5 Code Representation of Graph Concepts

In the previous sections, we saw basic modeling entities: node and relationship types. We have to clarify in what way those entities are represented inside the code itself. The representation of node types and node relationships inside the code depends on the programming language that we use. We can define quite distinct representations of those entities depending on the fact that we are in an object-oriented language or a functional language. However, we think the spirit of graph-oriented programming can be quite the same in all sorts of programming languages. For sure, as of now, there is no graph-oriented programming language in the market.

In the case of object-oriented languages, for sure, the native aggregation-composition patterns should not be used, because they create couplings (cf. part 1.1). The use of specialization is trickier: it should be used carefully in an idea of reusability without coupling generation[4]. For functional languages, the problem is the same: the representation of a node cannot contain pointers to other nodes, or we have created couplings.

We can define the minimum set of requirements that we need to ensure the proper working of a graph-oriented set of programs (see table 1).

With the requirements described in table 1, we can create nodes and relationships[5], set a relationship to point to known nodes (source and destination). In order to manipulate the graph from code outside of the various graph elements, we need a way to represent the graph as an opaque structure that will be accessible through a graph manipulation API[6]. We can propose a minimum list of requirements for the graph manipulation API (see table 2).

In terms of programming,each individual node and relationship is manipulated alone, while manipulations that require more than one element are performed in the graph structure itself through the graph manipulation API. This way of proceeding has many advantages because there is no temptation to tightly

[4] Note that some graph databases are proposing already inheritance of node types and inheritance of relationship types.

[5] By nodes and relationships, we mean node and relationship type instances.

[6] Application programming interface.

Requirement	Sample in OO language	Sample in functional language
Programmatic representation of node type	Class	List/struct with typed members
Programmatic representation of relationship type	Class with a source node ID attribute and a target node ID attribute	List/struct with typed members with two members for the source and target node IDs
Programmatic representation of graph	There must be a class `Graph` enabling graph manipulation.	There must be a structure representing the graph and enabling graph manipulation.
Graph manipulation API	Methods on the `Graph` class	Functions acting on the graph structure

Table 1. Minimum set of requirements for a graph-oriented implementation

#	Category	Requirement Description
01	Basic	Create graph from select query
02	Basic	Get graph root node (when applicable)
03	Basic	Add nodes and relationships inside the graph
04	Basic	Delete a node or a relationship inside the graph
05	Basic	Modify a node or a relationship inside the graph
06	Basic	Get nodes and relationships from the graph to access them in a object-oriented or functional way
07	Advanced	Assert a topology condition on the graph (returning `true` or `false`)
08	Advanced	Search for nodes and relationships with some criteria (such as per attribute value)
09	Advanced	Merge two different graphs
10	Advanced	Persist the graph
11	Advanced	Match a pattern in the graph
12	Advanced	Perform some other complex operations on graphs (for instance, for two graphs G_1 and G_2, create the graph $G_3 = G_1 \cap G_2$)

Table 2. Minimum set of requirements for a graph manipulation API

couple things together. Actually, the way the graph is implemented internally has no real importance on developments, which is why the programming language is not so important in that case.

The requirements are quite common for graph libraries, except perhaps the requirement #07 that can be use to check if the graph is in a certain state. Typically a small DSL can be of use in the API of this requirement, the development of it requiring most probably the use of graph isomorphisms algorithms.

2.6 Software Structures Nearer From Business Concepts

In enterprise software, most software engineers have realized for a long time that it was very useful to have an *business layer* that represented as accurately as possible the business concepts [7]. In a way, the success of object-oriented programming was to enable this new way of modeling applications. We saw in section 1.1 that the code and database representation of the business semantics was not optimal.

In the OOP/RDBMS paradigm, the first problem is *adapting the business objects to the database*, because in the case of the `0..1` or `0..n` couplings, the

aggregation/composition must be transformed in foreign keys is not automatic. It can be written by hand of delegated to an ORM, but the adaptation between the code structures and the database structures must be done. In the graph-oriented programming model, the code structures and their relationships are the exact image of the database[7]. In a way, in terms of data structures, we can say that the graph managed by the graph library is an optimal representation of the graph in the database.

The second big concern that remains is *the positioning of the business logic on classes*. All the software engineers that developed domain-driven enterprise software have quite often a hard time positioning the methods on the right classes in the graph of business classes. As we saw in Fig. 1, the method `methodA1` must know a part of the topology of the graph to be able to run. With a certain set of use cases, the positioning can be solved, but when the software evolves, it is often hard to redistribute the business logic to fit the new requirements.

The case of a change in the business semantics from $\overline{A} \rightarrow \overline{B}$ to $\overline{A} \rightarrow \overline{C} \rightarrow \overline{B}$ is typical: the evolution of `methodA1` calling `methodB1` is complex to realize. Will `methodB1` keep the same scope? Must we introduce a `methodC1` on `Class C` calling `methodB1` and called by `methodA1`? Will `methodC1` contain some business logic that was originally in `methodA1`? And we are here in a very simple case.

In the graph-oriented programmingparadigm, we can solve the business logic positioning through the use of graph transformations.

3 Graph Transformations

The notions of graph transformation, graph grammars, graph rewriting, are widely studied from the 70s in the domain of theoretical computer science and mathematics[8]. All those works are related to directed or undirected graphs, with ou without labels and colors. The works done around Progres and AGG [4] were particularly inspiring.

The objective of this section is the materialize the notion of graph transformation in the software itself and to use it to build enterprise software. We will show that the graph transformation structures are near from functional programming and that they enable to write programs that do not generate technical debt.

3.1 Modeling Graph Transformations

We propose the following definition of the graph transformation: A graph transformation is a function that takes in input a graph (or a subgraph) and gives in output a graph (or a subgraph). In object-oriented languages, the graph transformationcan be represented by a method of a (treatment) class. The output

[7] Some database vendors encourage also the use of native types to have no type conversion at all.

[8] For instance [16,13] amongst other references.

graph can be the same graph modified (destructive approach also called with side effects) or a new graph (no side effect approach).

The subgraph will be represented as a container because it contains node and relationship types. The graph transformation turns one subgraph (`INBOUND`) into another subgraph (`OUTBOUND`) like shown in Fig. 5)[9]. In a way, graph transformations are first-class citizens in the design (whereas they are not in UML).

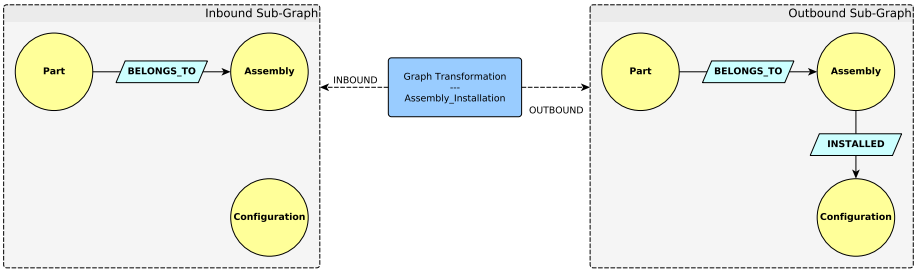

Fig. 5. Representation of graph transformation

3.2 The Graph Transformation Code Structure

A graph transformation must take in charge three problems: check the input subgraph topology to determine its applicability; transform the inbound subgraph into the outbound subgraph following business rules; analyze and solve rewiring issues at the limit of the subgraph.

The graph transformation must primarily check the input subgraph topology to see of the transformation is applicable. By checking the topology of the graph, the graph transformation determines what nodes and relationships it expects to find in the subgraph to be applicable. In case the topology is not compliant with the expected topology, the graph transformation should declare itself as `*NOT_APPLICABLE*`.

This check of topology must: be strictly limited to the required nodes and required relationships; not presume about other relationships that may exist for certain nodes and that are not relevant in the context of the graph transformation that is being executed; be expressed in a graph-oriented topology language enabling `assert`-like clauses.

The real graph transformation will take place after the topology check. It can modify the existing subgraph or create a new subgraph following the business rules of the business domain.

The challenge of the third step is to ensure that only relevant nodes and relationships were *seen* by the graph transformation. When the resulting subgraph is a part of a bigger graph, the modification must be done locally without

[9] The Henshin tool is proposing a graphical DSM to represent and control the validity of graph transformations [18].

damaging, by mistake, the relationships that are present but not relevant in the context of the graph transformation that is being executed.

This is a crucial point. It means that it is as if the graph transformation was acting on a *graph view* and was *blind* to whatever information that is not relevant to it. For instance, in Fig. 6, `h` is not modifying the inbound or outbound relationships of `N`. If `h` is correctly coded, it should not see any of those gray relationships and nodes.

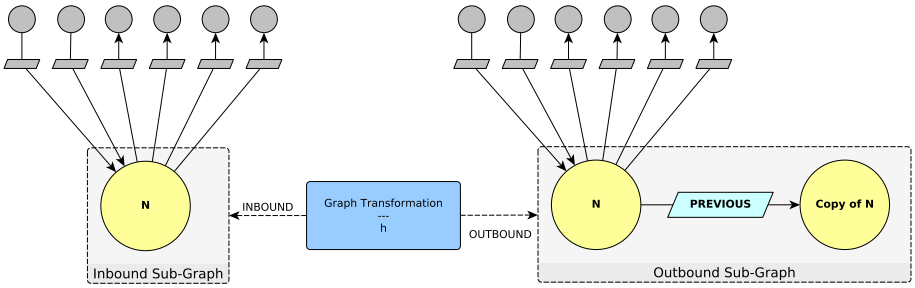

Fig. 6. Rewiring preserves unknown relationships

This is the guarantee of the *infinite and easy evolution of the software*: provided the graph transformation does not assume more than it *strictly needs to know*, and provided the graph transformation may not apply in case of topology mismatch, we have the foundations of an *ever-evolving system with no technical debt*.

3.3 What are Graph Transformations?

The graph transformation knows about the subgraph topology and potential nodes and relationships attributes. It can be seen as the *minimal coupling unit* in an absolute way: it only sees the view when the topology conditions are matched, acts on it without assuming more things about the "invisible" parts of the graph. In that sense, if a business rule is encoded into a graph transformation, we can say that the encoding of the rule is *optimal*.

We could see the graph transformation as *a kind of method of a graph*, the graph being *a complex object*. Actually, in a lot of cases, the object-oriented approach is not sufficient or fully adapted to represent properly (complex) business concepts, whereas graphs are. In terms of treatments, the graph transformations are attached to the graph as the methods are attached to the object (through the topology checks).

That is why graph-oriented programming can be interpreted as a natural evolution, or a generalization, of object-oriented programming: objects became (sub)graphs and methods became (sub)graph transformations.

3.4 Composing Graph Transformations

In current enterprise software, many programs are big and are taking in charge a lot of cases in one single piece of code. This code embeds various knowledge of data topology.

In graph-oriented programming, graph transformations are linked to their topology conditions, which means that, contrary to classical programming models, there should not be, inside a graph transformation code, business rules that apply to various topologies. In other words, graph transformations can be quite small, attached to a particular topology pattern. This way of proceeding enables bottom-up programming [12] for the implementation of business rules.

Composition of graph transformations are eased by their uniform interface: subgraph in inbound and subgraph in outbound. The fact that they protect themselves from non applicability enables their easy composition. As a matter of facts, graph transformations appear as the most reduced compose-able units of coupling in the graph-oriented design software. They are the main kind of enterprise software building blocks. Actually, composition makes reusability possible.

For instance the `h` transformation of Fig. 6 is creating a snapshot of any node `N`, `Copy of N` at a certain time and links it to `N` through an instance of the `PREVIOUS` relationship. The current `N` is preserved, so is all the relationships that are invisible to `h`. `h` is a reusable graph transformation that can be composed before or after the application of any `f` graph transformation to create respectively `f` ∘ `h` or `h` ∘ `f`.

3.5 Graph Transformation Evolution Rules

Graph transformations evolutions should follow some to avoid generating temporal couplings (cf. part 1.2), the objective being to determine what we must do in front of a change in business logic. We can propose the following rules.

If the topological applicability conditions change, then the graph transformation should most probably be forked, because this is a change in the applicability domain. If the topological conditions do not change, it depends on data time characteristics: if all data in the database are submitted to the new rule (and none to the previous version), the graph transformation can be updated; if old data still obey the previous version of the rule, the graph transformation should most probably be forked. We must note that a change in data structure (node and relationship typed members) must be considered a topological change.

Forking graph transformations enable to limit, as much as possible, the creation of temporal couplings, the objective being to have a code level representation of the business semantics that is not creating artificial technical couplings. Forking graph transformations also creates a timeline-based design for the application, where data structures and graph transformations are applicable or not depending on the time.

This implies that some other orchestration code will exist in order to find the proper graph transformation on two criteria: the business rule to apply plus

the time of the application. This way of building applications enable to create simply very complex applications in which both the data and the business rules can evolve with time. We could call this kind of software "timelined enterprise software".

4 Conclusion

We provided, in this article, a very first overview of the graph-oriented programming paradigm. We believe this paradigm enables to solve structural and temporal couplings issues. With the concepts that we have defined, we presented the building blocks to create a new generation of enterprise software that would not generate technical debt and in which design, programming and data storage would be the nearest ever from the business semantics.

Patterns and prototypes are being developed currently in this paradigm, in particular in the domain of aerospace maintenance, business domain where the graph-oriented programming approach can help tackling the complexity.

References

1. D. Alur. *Core J2EE Patterns*. Prentice-Hall, 2^{nd} edition, 2003.
2. F. Buschmann. *POSA Volume 1 - A System of Patterns*. Wiley, 1996.
3. W. Cunningham. The wycash portfolio management system, 1992. c2.com.
4. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2. World Scientific, 1999.
5. D. Allemang et al. *Semantic Web for the Working Ontologist*. Elsevier, 2008.
6. N. Brown et al. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, 2010.
7. E. Evans. *Domain-Driven Design*. Addison-Wesley, 2003.
8. M. Fowler. *Analysis Patterns, Reusable Object Models*. Addison-Wesley, 1996.
9. M. Fowler. *Refactoring*. Addison-Wesley, 1999.
10. M. Fowler. Technical debt, 2003. martinfowler.com.
11. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements Of Reusable Object Oriented Software*. Addison-Wesley, 1994.
12. P. Graham. *ANSI Common Lisp*. Prentice Hall, 1996.
13. U. Prange H. Ehrig, K. Ehrig and G. Taentzer. *Fundamentals of algebraic graph transformation*. Springer, 2010.
14. D. Longstreet. *Function Points Analysis*, 2008. softwaremetrics.com.
15. R. King R. Hull. Semantic database modeling. 1997.
16. G. Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation, Foundations*, volume 1. World Scientific, 1997.
17. David Sankel. Building software capital, 2016. YouTube CppCon2016.
18. T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer. Henshin: advanced concepts and tools for in-place EMF model transformations. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2010.