Graph Transformations and Software Engineering: Success Stories and Lost Chances - Highlights

Giovanni Toffetti, Mauro Pezzè

September 30, 2011

- Graph transformation systems are a powerful tool to deal with syntax, semantics and transformation of diagrammatic/visual notations
- Visual and diagrammatic languages are fundamental in software engineering but graph transformations are still not common practice in industry
- We look back at graph transformations in software engineering in the last fifteen years and identify success stories
- We discuss to what extent graph transformation succeeded, what are the main causes of failures, and how they can help software engineering in the next fifteen years

Graph Transformations and Software Engineering: Success Stories and Lost Chances

Giovanni Toffetti[a,*], Mauro Pezzè[b,c]

[a] *University College London*
[b] *University of Lugano*
[c] *University of Milano Bicocca*

Abstract

Textual as well as visual and diagrammatic notations are essential in software engineering, and are used in many different contexts. Chomsky grammars are the key tool to handle textual notations, and find many applications for textual languages. Visual and diagrammatic languages add spatial dimensions that reduce the applicability of textual grammars and call for new tools.

Graph transformation systems have been studied for over forty years and are a powerful tool to deal with syntax, semantics and transformation of diagrammatic notations. The enormous importance of visual and diagrammatic languages and the strong support that graph transformation provide to the manipulation of diagrammatic notations would suggest a big success of graph transformation in software engineering.

In this paper we discuss the main features of graph transformation and how they can help software engineers. We look back to the many attempts to use graph transformations in software engineering in the last fifteen years, identify some success stories, and discuss to what extent graph transformation succeeded, when they have not succeeded yet, what are the main causes of failures, and how they can help software engineering in the next fifteen years.

Keywords:
Graph transformation, software engineering, model, model transformation, visual language, dynamic system, evolving system.

*Corresponding author
Email addresses: g.toffetti@ee.ucl.ac.uk (Giovanni Toffetti), mauro.pezze@usi.ch (Mauro Pezzè)

1. Introduction

Informatics is about information and its representation. Both scientists and practitioners use many textual and diagrammatic notations to represent any kind of concept, from data to *models*, all the way to *programs* and computation. Understanding and defining notations is essential to use them properly, and *formal languages* are the best means to avoid ambiguities and misinterpretations, and to enable automatic information processing. Formal string grammars that Chomsky studied since the fifties are the most common way to specify the infinite set of all valid statements (the syntax) of textual (sequential) languages. As such, to name but one important application, they play a fundamental role in compilers.

Diagrammatic notations introduce new dimensions beyond the sequential successor relation that characterizes textual notations, and classic Chomsky grammars cannot capture these new dimensions. In the context of diagrammatic representations, the simplest and most natural abstraction is a graph, and graph grammars, or more generally *graph transformation*, are the most suitable specification for diagrammatic languages (graphs). As stated by Heckel: "Graph transformation has originally evolved in reaction to shortcomings in the expressiveness of classical approaches to rewriting, like Chomsky grammars and term rewriting, to deal with non-linear structures" [1].

Graphs are particularly suited for modeling and representing many structured and dynamic contexts, for instance software architectures, distributed communications, call graphs and many more. Graph transformation systems are not only an intuitive way to represent the syntax of graphs, but also to formalize how graphs evolve. As stated by Ehrig et al.: "Graph transformation allows one to model the dynamics in all these descriptions, since it can describe the evolution of graphical structures" [2].

Research on graph transformation started over forty years ago, and right from its inception it identified the main application directions. In 1969, Pfaltz et al. gave the first examples of classes of graph grammars (called Web grammars), setting the path to using graph grammars as formal specifications of diagrammatic languages [3]. In 1971, Pratt adxdressed string to graph language mappings with pair grammars, paving the way to the coming generalization of pair grammars into triple grammars [4] and their application to model transformation [5]. Finally, in 1973, Ehrig et al. introduced

the algebraic approach to graph transformation, which has proven a fundamental instrument in studying and demonstrating several properties of graph transformation systems [6].

Further application areas and methodologies were identified early on, both applying graph transformation in a generative and in an analytic approach. For example in the eighties, Dolado and Torrealda used a generative approach to generate Forrester diagrams [7], while Göttler proposed a generative approach to automatically derive diagram editors from graph transformation [8], becoming a cornerstone for many more approaches to develop domain-specific languages together with their editors and CASE tools. On the other end, Bunke proposed an analytic application of graph transformation to "read" diagrammatic notations (interpreting diagrams and flowcharts) [9].

The nineties saw the interest in graph transformations spreading to other communities that applied graph transformation in different fields, like software architectures [10, 11] and reconfiguration for fault tolerance [12] to name a few.

Over forty years of theoretical and applied research have shown the efficacy and the limits of graph transformation systems. Graph trasformation systems are an essential tool to formally define diagrammatic languages, find many applications in informatics, and are widely accepted in the academic community. However, notwithstanding a constant effort from the graph transformation community to promote industrial acceptance with dedicated workshops and symposia, for example the series: Graph-Grammars and Their Application to Computer Science, widespread application of graph transformation in common practice is still lacking.

In this paper we discuss the impact of graph transformation in software engineering. We do not aim to survey all the relevant applications of graph transformation to software engineering, but we focus on identifying the way graph transformation can be used in software engineering looking at a set of key application areas. We illustrate the different way of using graph transformation by discussing some sample work and analyzing the key elements of success and the reasons behind the lack of wide adoption so far.

The next section introduces graph transformation to make the paper self contained. Section 3 discusses the role that graph transformation has played in software engineering. Section 4 illustrates the suitability of graph transformation in software engineering by presenting few sample cases. Section 5 discusses the impact of graph transformation in academia and in industry.

Section 6 concludes and discusses the future of graph transformation in software engineering.

2. Graph transformation in a nutshell

Graph transformations have be defined in many ways with different notations to represent the rules, matching and glueing models, application conditions and more. In this section we summarize the main elements of graph transformation referring to the intuitive presentation of Heckel [1]. The interested readers can find a clear introduction and classification of graph transformation systems in the paper by Blostein et al. [13] and a complete and formal definition in the "Handbook of graph grammars and computing by graph transformation" by Rozenberg and Ehrig [14].

Graph transformation defines transformations over graphs by means of rules. A graph is a set of nodes and (directed or undirected) edges. Both nodes and edges can be typed to represent different concepts, for example a person or an object, and can have attributes, defined as name and value pairs [1]. The *type* of nodes and edges of a graph can be specified intuitively by means of sets of *terminal* and *non-terminal* nodes and edges. A more expressive way to define types is by means of another graph, called *type graph*. A type graph is a graph that represents the types of nodes and edges, their attributes, and what is *valid* in an *instance* graph based solely on type information and cardinality. For instance a node of type *person* can only be connected by an edge of type *is natural child of* to two instances of the node of type person. A type graph typically contains a single instance of node per type, while edges represent both edge types, the types of nodes they can connect, and the allowed cardinality. Montanari et. al give a clear and practical example of type graphs applied to Entity-Relationship modelling [15].

Once the types of nodes and edges are defined, we can define the *rules*, also called productions, that represent the core of graph transformation systems. A rule is composed of a left-hand side (LHS) and a right-hand side (RHS). The LHS indicates the pre-conditions for applying the rule, while the RHS indicates the post-conditions. A rule can be applied to a graph when the LHS *matches* a subgraph, and the application of the rule replaces the matched subgraph with the RHS of the rule.

Figure 1 shows an example of rule (top of the figure) taken from [1]. The LHS of the rule is the graph L and the RHS is the graph R. The bottom

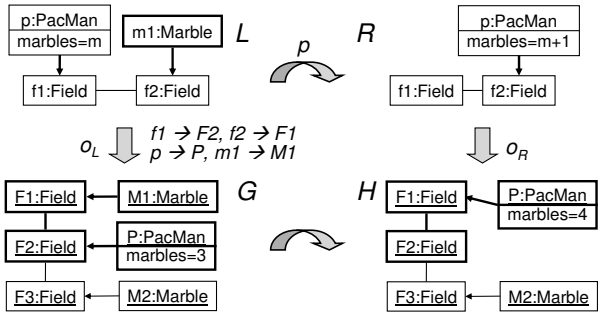

Figure 1: Rule and application from Heckel [1]

part of the figure shows the application of the rule to transform the graph G into H. The matching elements of LHS and G are represented by the mapping O_L, where O stands for "occurrence". The rule is applied in three main steps: match the LHS to a subgraph of G, modify G by deleting all the elements of the matched subgraph that are not preserved in RHS (i.e., LHS \ RHS), modify G by adding the new elements introduced by RHS (i.e., RHS \ LHS) obtaining H.

Deleting a subgraph (step 2) may leave dangling edges, that are edges not connected to two nodes, and hence produce invalid graphs. A common solution to this problem is augmenting the rules with a graph K that indicates elements that must exist in the graph to apply the rule and are preserved by the application of the rule, thus preventing edges to remain partially disconnected during the application of the rule. The graph K is called the *gluing condition* and the approach is called double-pushout approach (DPO) [6].

Rules can be applied *directly*, when the users specify the rule to apply and the matching to be used, or in *undirected* fashion, when rules are ordered or organized in layers and applied repeatedly until no further matchings are possible.

To expressiveness of graph transformation has been enhanced with several extensions, the most common being *constraints* and (negative) *application conditions*. Application conditions (ACs) and *negative* application conditions (NACs) limit the application of rules to guarantee that some required properties are satisfied. They specify the *context* in the graph that must (AC) or *must not* (NAC) be matched. ACs and NACs are powerful constraints at rule level that allow complex specifications, for example a user

can be *member* of at most one role, but can be *authorized* for a role if the role is inherited from the role of assignment. The key characteristics of application conditions is that the context to which they refer to can be arbitrarily far from the nodes to which the rule is applied, allowing the designer of the rule to specify elaborated conditions. Some popular tools, like PROGRESS, further enhance the expressive power of rules and conditions by *set nodes* and *paths*. Set nodes, also called multi-objects, are used as a short-hand notation to represent a set of rules in which the set node appears n times with $n \geq 0$. This allows to specify for example rules deleting all instances of a give node type. Paths indicate sequences of directed edges of arbitrary length, and allow to represent concepts such as for instance multi-hop connectivity in graphs.

3. Graph transformation in software engineering

Graphs are a natural *visual* representation of structured information, With nodes modeling entities and (hyper) edges (n-ary) relationships among entities, they provide a simple visual abstraction of the reality. In particular, they provide a powerful means to represent nodes and edges with types, attributes, and composition constraints.

In software engineering, modeling is a basic tool used at all abstraction levels and during the whole software development process. Models and in particular diagrammatic models based on graphs are used to represent and analyze software procedures, processes, components, functioning, architectures, interactions, as well as the entire software life cycle in its various phases and declinations, like re-factoring, testing, bug assignments, deployment and composition. Thus, graph transformation systems have a natural application in several areas of software engineering.

Software engineers have exploited three main features of graph transformation systems: their ability to provide an intuitive diagrammatic representation of modeling concepts, their suitability to formally specify graph languages and mechanisms to guarantee that (or verify whether) a graph is a valid language production and their flexibility in modeling and reasoning about graph evolution, as a representation of dynamic behavior, with rules and the possibility to prove dynamic properties, for example by studying parallel/sequential rule applications. So far, graph transformation systems have been applied in various contexts:

modeling several aspects of software and the software development process to prove, classify, reason over their characteristics;

defining visual languages and their tools to produce domain-specific development environments, with focus on the syntax, the specification of the visual language;

modeling transformations and mappings across notations and languages;

checking and proving the consistency of dynamic systems with focus on the semantics, the dynamics of the representation.

The paper by Blostein et al. provides a thorough analysis of the state of the practical use of graph rewriting systems up to 1996 [13]. At that time, despite the advantages in terms of abstraction, correctness, and convergence, graph transformation systems had not attained widespread practical use. Blostein et al. identified as a factor hindering the industrial application of graph transformation in software engineering the lack of education and tools with consequent lack of experimental data about the advantages of graph transformation in the field. In the last fifteen years, the graph transformation community has endured a consistent effort, producing a vast amount of educational material, a series of conferences dedicated to practical uses of graph transformation, as well as a set of development tools that have moved from research prototypes to products used in industrial scale projects, like *Fujaba*.

Blonstein et al. identified also lack of graph rewriting system modularization, grammar evolution, and graph inspections support as key technical factors hindering the industrial applicability of graph transformation. These problems are grounded in the lack of design techniques and patterns, efficiency, appropriate choice of rule organization (i.e., graph grammars, unordered, ordered, and event-based graph rewriting systems), structure of large rule collections and common extensions to rewrite mechanisms (for instance application conditions and set-rules). Notwithstanding the research effort, some of the considerations of Blonstein et al. are still valid today, and are affecting the industrial success of graph transformation.

In the following section, we sample some applications of graph transformation to software engineering, highlighting how different features of graph transformation have exploited in different contexts and discussing the limits of the solutions.

4. Application examples

The many applications of graph transformation in software engineering exploit the ability of modeling both static and dynamic aspects of diagrammatic representation, the possibility of reasoning about model evolution and the ability of supporting visual languages and their semantics.

In this section we present some examples of applications that exploit the different features of graph transformation. The examples represent applications in different fields and aim to indicate how graph transformation can meet different needs, without claiming neither completeness nor generality.

4.1. Modeling static and dynamic aspects

While capturing the syntactic aspects of diagrammatic notations is quite simple, formally modeling and reasoning about the dynamic semantics of diagrammatic notations is difficult. Graph transformation represents well both static and dynamic aspects, as well as the interplay between syntax and semantics, and provides a powerful means to analyze the diagrams and their evolution.

In this section we illustrate the ability of modeling both static and dynamic aspects referring to applications of graph transformation to software architecture styles and security.

Software Architecture Styles

Software architecture styles constrain the freedom of software architects who shall follow specific conventions in identifying and composing subsystems and components. Designing architectures compliant with a given style requires the ability to enforce some rules that often reflect dynamically evolving constraints on diagrams. Graph transformation is an ideal means to define composition rules that can be dynamically adapted to new needs and requirements, and can be used to derive syntax driven editors, analyze the compliance of architecture designs with a given style and define transformations to change architectural style.

To represent and verify software architecture styles we need both a formal framework and a *natural* (graphical) way to express software design choices, features that characterize graph transformation. Le Metayer suggested to define *entities* (like procedures, modules and processes) as graph nodes, *communication links* as edges, and the constraints that characterize the *architecture style* as graph transformation rules [11].

In Le Metayer's approach, entities evolve by changing internal and public variables according to the semantics of the programming language, while a *coordinator* manages the architecture itself and maintains the consistency of the architecture style by adding and removing nodes and edges using conditional graph rewriting based on the values of public variables of the entities.

Graph transformation rules ensure the topological properties of the architecture *by construction*, and reconcile the *dynamic view* of the architecture, defined as its evolution through the coordinator rewrite system, with the *static verification* of the architectural style defined on a set of graph transformation rules. The intuitiveness and formality of the notation, as well as the separation of the logic of the coordinator and the entities, support the definition and evaluation of properties of information flows for different architectural styles.

Graph transformation rules support the static type checking of the coordinator and the identification of violations of the constraints that characterize the architectural style. The type checker first builds a *reduction graph*, that is a cyclic graph whose origin is the left hand side of a coordinator rule, the terminal node is the axiom of the grammar associated to the architectural style, and the internal nodes are derived by applying the reverse grammar of the style. It then checks whether the right hand side of the coordination rule in all its application contexts that are given by the reduction graph can be reduced to the axiom of the architectural style grammar.

Le Metayer clearly indicates the advantages of graph transformation for enforcing architectural styles and indicates in the choice of context-free grammars the strength as well as the limitations of the approach: context-free grammars reduce the complexity of the checks, but limit the rules that can be defined.

Role-based access control

Access control (AC) is a common security mechanism used to determine and enforce which entities (programs or users) can have access to objects, for example files or devices, and with what permissions, for example read, write or execute. *Role-based access control* (RBAC) uses the notion of *roles* assigned to users within an organization to eliminate or at least reduce the errors in AC managers. RBAC are defined as constraints over roles, role evolution and resource accesses. The set of constraints and the dynamics of RBAC can be extremely complex, and can evolve over time following changes

in roles and access control policies. Consistency and completeness of access control rules are very important to prevent security leaks, but verifying that a complex set of evolving rules is consistent and complete is a hard problem. Koch et al. suggest that graph transformation can define formally and intuitively a set of RBAC constraints and can support automatic verification of important security properties [16]. Koch's framework provides an intuitive visual description of AC in terms of graphs, an expressive specification language for different AC schemas, the specification of static and dynamic consistency conditions through graphs, and an executable specification that can leverage on existing tools to verify the properties of a graph-based RBAC description.

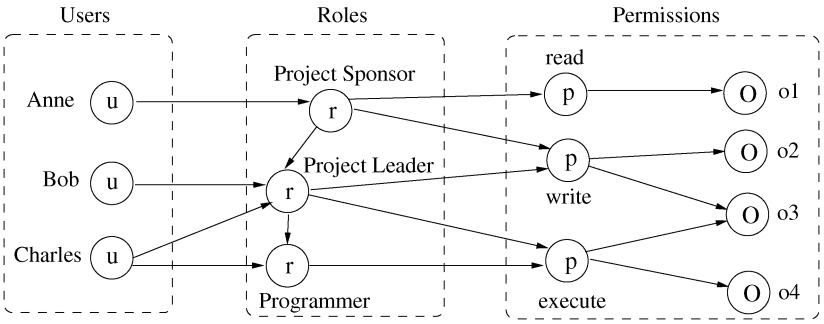

Figure 2: A graphical specification of user-role, role-permission and permission-object assignment from [16]

The approach relies on a specific *type graph* as a first constraint for graphs representing RBAC. As shown in Figure 2, the type graph models concepts such as *users* that can be associated to *roles*, and have *sessions*. Roles are associated to *permissions* on a set of objects. Hierarchies of roles are modeled with the semantics of (reverse) inheritance of permissions: For example a higher level role has a superset of the permissions of the lower level role.

Type graphs can model common RBAC concepts and schemas, and support the proof of the correctness of a graph-based RBAC specification with respect to common consistency requirements, for example dynamic separation of duties. can be specified as *graphical constraints* that are expressed as undesired subsystem states, and can be automatically verified.

4.2. Model evolution

The scale of development and ubiquity of software increasingly require methodologies to assess and enforce software quality. With the aim of improving the software development process across all its phases, modeling techniques are being applied to represent and reason about several aspects and concerns, all the way from formal requirement collection, platform specifications, deployment, to collaboration diagrams. Maintaining consistency across heterogeneous models at different abstraction levels is a difficult task.

Model transformation systems represent a natural approach for dealing with multi models frameworks and are increasingly applied in this context to automatically generate code from models, maintain the consistency across several notations when updating a model, map informal onto formal representations to be used for proving properties, reflect changes to the formal representation into the informal one.

The most relevant contribution of graph transformation to model transformation are triple graph grammars (TGGs). They are a generalization of Pratt's pair grammars [5] and have been introduced by Schürr in 1994 [4]. TGGs aim to provide a mapping between different graph that represent different modelling languages and notations and that can be used to automate model transformations. They are called *triple* because their rules consist (generally on both LHS and RHS) of elements from three different (type) graphs: the *source*, the *target*, and the *correspondence* graphs. Figure 3 shows an example of a triple graph grammar diagram. The source and target graphs represent the notations to be mapped, while the correspondence graph acts as an explicit representation of the correspondence relations between elements of the source and target graphs. Any TGG can be compiled into a pair of forward and backward graph translations (respectively FGT and BGT) that can then be used to transform a graph in the source language into a graph in the target language (FGT) and back (BGT). These features make them extremely appropriate to specify and implement mappings across software engineering models and artifacts.

An interesting application of model transformation is the need of mixing informal and formal notations. The long debate between supporters of formal versus informal models indicate that we need both the ability of proving properties which is typical of formal models and the flexibility that is typical of informal models. Baresi and Pezzé introduced an interesting application of pairs of graph transformation systems to support the on the fly generation of formal notations from evolving informal diagrams to conjugate flexibility

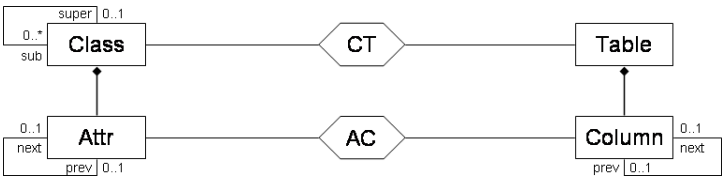

Figure 3: TGG schema that defines a correspondence structure (center) between elements of class diagrams (left) and relational database schemata (right) from [17]

and formality [18]. Practitioners can bend the semantics of informal diagrammatic notations by modifying simply graph transformation rules that adapt the formal models to the next interpretation.

As discussed in a recent paper by Schürr and Klar, TGGs found many practical applications, the most notable one is in the OMG's model transformation language QVT that adopted some fundamental ideas from TGGs [17]. Inefficient parsing, negative application condition semantics, lack of modularization, refinement, and reuse are still hindering the wide-spread application of TGGs.

4.3. Visual languages

Visual languages and in particular diagrammatic notations have been extensively used to represent complex models. The impossibility of matching the many needs of different application domains has triggered the interest in *domain-specific languages*, that are visual notations or full-fledged visual programming languages that, by concentrating on a clearly identified domain and its underlying assumptions, manage to offer representations that are at the same time very compact and expressive for a specific problem [19].

Domain specific languages are applied in niche areas sometime too small to justify the big effort required to design the visual language, define its semantics, and implement the associated editing tools. Providentially, "the syntax and static semantics of a visual language can be unambiguously defined using graph transformation" [20], and indeed several applications of graph transformations with this purpose are common in literature. An extensive and rigorous exposition on the subject can be found in the work of Bardohl et al. [21].

The common definition of a visual language separates the *concrete* from the *abstract* syntax of the language, as depicted in Figure 4. The concrete

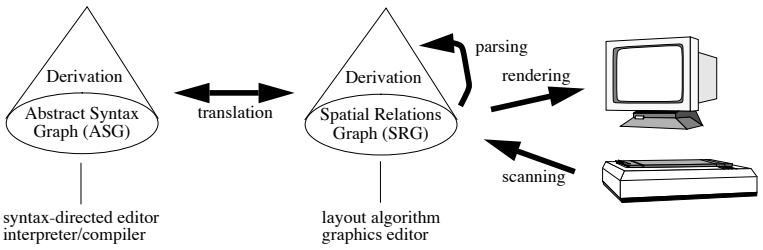

Figure 4: Abstract and concrete syntax of visual languages from [21]

syntax represents the diagrammatic symbols, for example boxes, bubbles and arrows, and the spatial relations, for example containment, connection and positioning, that are used in the visual notation. The concrete syntax of a visual language is usually given by means of a *spatial relationship graph* (SRG). The abstract syntax predicates on the concepts represented by the concrete symbols and relations, for example classes and relationships in a class diagram, and is commonly given as an *abstract syntax graph* (ASG). Figure 5 shows an example of a visual language rule representing both the abstract syntax (top) and the concrete one (bottom).

Graph transformation systems can naturally specify both the SRGs and ASGs, and thus support the generation of parsers for newly defined visual languages. Bardohl et al. write "Despite the wide-spread usage of visual modeling ... there is a considerable *lack of formalisms* for defining their syntax and semantics. ... As a consequence, most published visual languages come with informal an imprecise definitions, and the development of their tools often requires far too many person years" [21]. Graph transformation can solve the problem and boost the use of domain specific visual languages.

The most popular editors generated from graph transformation systems support either *syntax directed* or *free-hand editing*. Syntax directed editors offer a set of editing operations that designers can use to create diagrams, while free hand editors provide parsers for diagrams that users can edit freely with any editing tool. The complexity of building efficient diagram parsers privileges syntax directed over free hand editors. Syntax driven editors can be either *grammar-* or *transformation-based*. Grammar-based editors specify the language independently from the editor using a grammar, while in the transformation-based approach the language is defined by its editing rules. Grammar-based approach provide a more concise syntax definition, while

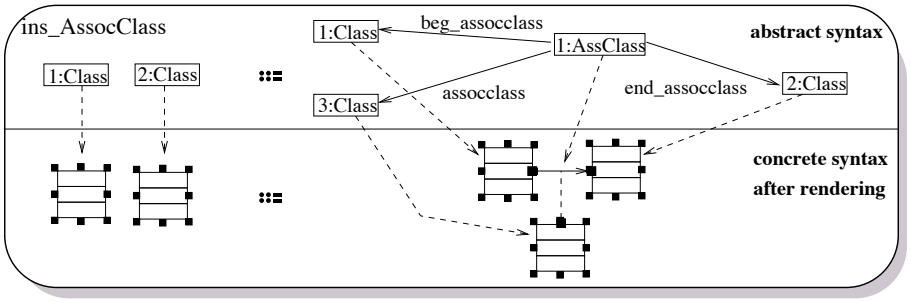

Figure 5: Rule describing the insertion of an association class (dashed arrows represent the coupling between abstract and concrete syntax) from [21]

transformation-based editors offer the possibility of modelling intermediate editing steps that might be invalid in the visual language syntax but are often needed to simplify the editing process, hence is the preferred solution.

Recently, graph transformation approached have been combined with meta-modelling solutions to specify visual language tools. An example is provided in De Lara et al. [22].

5. Academic and industrial impact

In the former chapter we sampled several applications of graph transformation that have been proposed in the last 15 years. The wide range of application fields and studies indicate that the scientific community believes in the possibility of overcoming the problems highlighted by Blostein et al. in 1996. Most of the work surveyed in the former chapter is still part of research projects, but there are some interesting industrial success stories that indicate how graph transformation can move from research project to industrial best practice.

One for all, we would like to mention the large and successful project on mechatronics carried on in the University of Paderborn that aims to develop a new generation transportation system in which unmanned, independent and lightweight trains combine each other into convoys to optimize the service offered to passengers [23]. The ambitious project has moved in a decade from visionary research to pre-competitive industrial prototype, and needs to deal with real time constraints in a hybrid embedded systems that shall evolve and self-adapt to different traffic and environmental condition. The

project relies on graph transformation to describe and analyze combination of heterogeneous components, evolution and self-adapting behavior [24]. As a side effect, the project has produced Fujaba, a CASE tool that support model-based software engineering and re-engineering and is based on graph transformation. The tool is now available for open use and is being applied to new industrial scale projects.

Software ubiquity is driving every day more the need of methodologies to assess and enforce software quality. As modelling techniques and guidelines continuously gain wider acceptance in the industrial practice, graph transformations are directly applied to study program behaviours discovery and verification [25], while graph grammars concepts have a major impact in model transformation and mapping (e.g., model to model, or code generation). Important results are achieved by combining graph transformation, programming languages, and mathematical modelling tools. The mutual effects that these different approaches have on each other contribute in advancing and improving the available tools and techniques. One notable example is the impact of triple graph grammars on OMG's model transformation language QVT [17]. But also graph transformations overcome some limitations by adopting external concepts. For instance, Legros et al. use graph rewriting to enforce modelling guidelines for automotive industry, and, in order to reduce the number of rules in common scenarios, extend graph transformation modularity by adopting programming language concepts such as *generics* and *reflection* [26].

Both academic and industrial experience indicate that graph transformation systems are the right tool to deal with diagrammatic models of dynamically evolving systems.

The recent years are seeing an increasing impact of systems that because of their nature and complexity cannot be designed and sealed at design time, but change and evolve at runtime. Notable examples are self-organizing or ad-hoc networks that have a potentially unbounded number of nodes, no centralized management, and evolve over time as nodes join or leave [27]. Another emerging area with highly dynamic and evolving characteristics is the area of autonomic and self adaptive systems. These systems are designed to adapt at runtime to different and evolving execution conditions. A particularly interesting area where graph transformations can be applied is the study of the behaviour and interaction of several concurrent control loops in autonomic systems [28].

6. Conclusions

In this paper, we sampled the applications of graph transformation systems to research and industrial projects to identify success stories and failure factors. Although over forty years of research and applications have largely advanced the state of art and practice of graph transformation, the key ingredients for the success of graph transformations have been already identified and studied in the early years.

Blostein et al. in a very thorough paper of fifteen years ago indicated education and technical factors that hindered the industrial success of graph transformation. In this paper we sample the research an industrial experience of the last fifteen years to identify advances and open problems. Although some of the key technical problems identified by Blostein, in particular lack of modularization and structure, are not completely solved yet, the projects surveyed in this paper indicate interesting advances in tool support and design techniques and patterns. The many research experiences and the few relevant industrial scale projects produced tools and methodologies that are ready for industrial use.

The research and industrial experience referred to in the former sections indicates that technical factors are not a major impediment to the use of graph transformation. The importance of mastering dynamically evolving systems at runtime is becoming a necessity in many relevant application domains like distributed systems and autonomic computing, and is opening new opportunities for graph transformation systems.

The industrial success of graph transformation does not depend on technical issues, but on commercial aspects, like the timeliness of success stories, the definition of a strong market for tools based on graph transformation systems, and the willingness to invest in formal approaches.

References

[1] R. Heckel, Graph transformation in a nutshell, Electron. Notes Theor. Comput. Sci. 148 (2006) 187–198.

[2] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer, Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series), Springer, 2006.

[3] J. Pfaltz, A. Rosenfeld, Web grammars, in: Int. Joint Conference on Artificial Intelligence, 1969, pp. 609–619.

[4] A. Schürr, Specification of graph translators with triple graph grammars., in: WG'94, 1994, pp. 151–163.

[5] T. Pratt, Pair grammars, graph languages and string-to-graph translations*, Journal of Computer and System Sciences 5 (6) (1971) 560–595.

[6] H. Ehrig, M. Pfender, H. Schneider, Graph-grammars: An algebraic approach, in: 14th Annual Symposium on Switching and Automata Theory, IEEE, 1973, pp. 167–180.

[7] J. Dolado, F. Torrealdea, Formal manipulation of Forrester diagrams by graph grammars, Systems, Man and Cybernetics, IEEE Transactions on 18 (6) (1988) 981–996.

[8] H. Göttler, Graph grammars and diagram editing, in: Graph-Grammars and Their Application to Computer Science, Springer, 1987, pp. 216–231.

[9] H. Bunke, Attributed programmed graph grammars and their application to schematic diagram interpretation, Pattern Analysis and Machine Intelligence, IEEE Transactions on (6) (1982) 574–582.

[10] T. Dean, J. Cordy, A syntactic theory of software architecture, Software Engineering, IEEE Transactions on 21 (4) (1995) 302–313.

[11] D. L. Métayer, Describing software architecture styles using graph grammars, IEEE Trans. Software Eng. 24 (7) (1998) 521–533.

[12] M. Derk, L. DeBrunner, Reconfiguration for fault tolerance using graph grammars, ACM Transactions on Computer Systems (TOCS) 16 (1) (1998) 41–54.

[13] D. Blostein, H. Fahmy, A. Grbavec, Issues in the practical use of graph rewriting, in: Graph Grammars and Their Application to Computer Science, Springer, 1996, pp. 38–55.

[14] G. Rozenberg, H. Ehrig, Handbook of graph grammars and computing by graph transformation, Vol. 1, World Scientific, 1997.

[15] A. Corradini, H. Ehrig, M. Löwe, U. Montanari, J. Padberg, The category of typed graph grammars and its adjunctions with categories, in: J. E. Cuny, H. Ehrig, G. Engels, G. Rozenberg (Eds.), TAGT, Vol. 1073 of Lecture Notes in Computer Science, Springer, 1994, pp. 56–74.

[16] M. Koch, L. V. Mancini, F. Parisi-Presicce, A graph-based formalism for RBAC, ACM Trans. Inf. Syst. Secur. 5 (2002) 332–365.

[17] A. Schürr, F. Klar, 15 years of triple graph grammars, Graph Transformations (2008) 411–425.

[18] L. Baresi, M. Pezzè, Formal interpreters for diagram notations, ACM Trans. Softw. Eng. Methodol. 14 (1) (2005) 42–84.

[19] H. Göttler, Diagram editors = graphs + attributes + graph grammars, International Journal of Man-Machine Studies 37 (4) (1992) 481–502.

[20] D. Blostein, A. Schürr, Computing with graphs and graph transformations, Softw., Pract. Exper. 29 (3) (1999) 197–217.

[21] R. Bardohl, G. Taentzer, M. Minas, A. Schürr, Application of graph transformation to visual languages, World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999, pp. 105–180.

[22] J. de Lara, H. Vangheluwe, Defining visual notations and their manipulation through meta-modelling and graph transformation, Journal of Visual Languages and Computing 15 (3-4) (2004) 309 – 330, domain-Specific Modeling with Visual Languages.

[23] C. Henke, M. Tichy, T. Schneider, J. Böcker, W. Schäfer, System architecture and risk management for autonomous railway convoys, in: Proc. 2nd Annual IEEE Intl. Systems Conf, 2008.

[24] T. Eckardt, C. Heinzemann, S. Henkler, M. Hirsch, C. Priesterjahn, W. Schäfer, Modeling and verifying dynamic communication structures based on graph transformations, Computer Science-Research and Development (2011) 1–20.

[25] C. Zhao, J. Kong, K. Zhang, Program behavior discovery and verification: A graph grammar approach, IEEE Transactions on Software Engineering (2010) 431–448.

[26] E. Legros, C. Amelunxen, F. Klar, A. Schürr, Generic and reflective graph transformations for checking and enforcement of modeling guidelines, J. Vis. Lang. Comput. 20 (2009) 252–268.

[27] M. Saksena, O. Wibling, B. Jonsson, Graph grammar modeling and verification of ad hoc routing protocols, in: Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems, Springer-Verlag, 2008, pp. 18–32.

[28] Y. Brun, G. Di Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, M. Shaw, Engineering self-adaptive systems through feedback loops, Software Engineering for Self-Adaptive Systems (2009) 48–70.